ISO TC 22/SC 3 N

Date: 2002-11-15

ISO/CD 14229-1.4

ISO TC 22/SC 3/WG 1

Secretariat: FAKRA

Road vehicles — Diagnostic services — Part 1: Specification and requirements

Élément introductif — Élément central — Partie 1: Élément complémentaire

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: International Standard
Document subtype:
Document stage: (30) Committee
Document language: E

C:\Documents and Settings\E18221\My Documents\Current Work\Reversaid\Calibration Tool Development References\ISO CAN specs\ISO DOCS\ISO 14229-1_(E).doc STD Version 2.1

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 14229-1 was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 3, *Electrical and electronic equipment*.

This second/third/... edition cancels and replaces the first/second/... edition (ISO 14229:2000), [clause(s) / subclause(s) / table(s) / figure(s) / annex(es)] of which [has / have] been technically revised.

ISO 14229 consists of the following parts, under the general title *Road vehicles — Diagnostic services*:

— *Part 1: Specification and requirements*

Introduction

This International Standard has been established in order to define common requirements for diagnostic systems, whatever the serial data link is.

To achieve this, the standard is based on the Open Systems Interconnection (O.S.I.) Basic Reference Model in accordance with ISO 7498 and ISO/IEC 10731, which structures communication systems into seven layers. When mapped on this model, the services used by a diagnostic tester (client) and an Electronic Control Unit (ECU, server) are broken into:

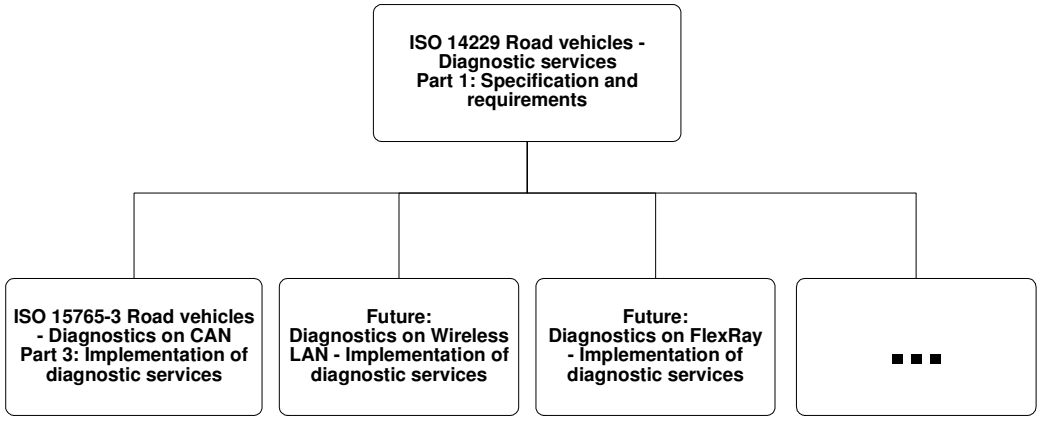
— Diagnostic services (layer 7),

— Communication services (layers 1 to 6).

NOTE The diagnostic services in this standard are implemented in various applications e.g. Road vehicles – Tachograph systems, Road vehicles – Interchange of digital information on electrical connections between towing and towed vehicles, Road vehicles – Diagnostic systems, etc. It is required that future modifications to this standard shall provide long-term backward compatibility with the implementation standards as described above.

Applicability	OSI 7 layer	Enhanced diagnostics services (non emission-related)	
Seven layer according to ISO/IEC 7498 and ISO/IEC 10731	Application (layer 7)	ISO 14229-1/ISO 15765-3	ISO 14229-1/other standards
	Presentation (layer 6)	---	---
	Session (layer 5)	ISO 15765-3	---
	Transport (layer 4)	ISO 15765-2	---
	Network (layer 3)	ISO 15765-2	---
	Data link (layer 2)	ISO 11898	---
	Physical (layer 1)	ISO 11898	---

The following figure shows an example about the possible future implementation of ISO/CD 14229-1.4 onto various data links.



Road vehicles — Diagnostic services — Part 1: Specification and requirements

1 Scope

This International Standard specifies data link independent requirements of diagnostic services, which allow a diagnostic tester (client) to control diagnostic functions in an on-vehicle Electronic Control Unit (server) such as an electronic fuel injection, automatic gear box, anti-lock braking system, etc. connected on a serial data link embedded in a road vehicle.

It specifies generic services, which allow the diagnostic tester (client) to stop or to resume non-diagnostic message transmission on the data link.

This standard does not apply to non-diagnostic message transmission, use of the communication data link between two Electronic Control Units.

This standard does not specify any implementation requirements.

The vehicle diagnostic architecture of this standard applies to:

— a single tester (client) that may be temporarily or permanently connected to the on-vehicle diagnostic data link, and

— several on-vehicle Electronic Control Units (servers) connected directly or indirectly.

See figure below.

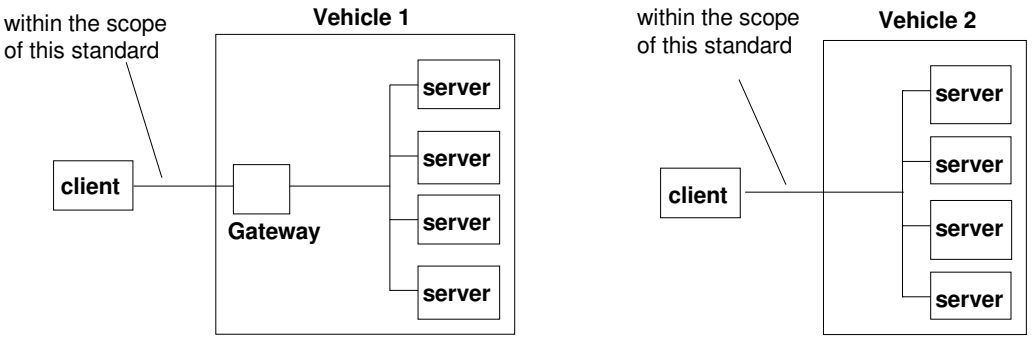


Figure 1 — Vehicle diagnostic architecture

Figure description:

— In vehicle 1, the servers are connected over an internal data link and indirectly connected to the diagnostic data link through a gateway. This document applies to the diagnostic communications over the diagnostic data link; the diagnostic communications over the internal data link may conform to this document or to another protocol.

— In vehicle 2, the servers are directly connected to the diagnostic data link.

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.

2.1 ISO publications

ISO 7498-1:1984	*Information processing systems - Open systems interconnection - Basic reference model*
ISO TR 8509:1987	Information processing systems - Open systems interconnection - Conventions of services
ISO 4092:1988/Cor.1:1991	Road vehicles - Testers for motor vehicles - Vocabulary Technical Corrigendum 1
ISO 15031-2	Road vehicles — Communication between vehicle and external equipment for emissions-related diagnostics — Part 2: Terms, definitions, abbreviations, and acronyms
ISO 15031-5	Road vehicles — Communication between vehicle and external equipment for emissions-related diagnostics — Part 5: Emissions-related diagnostic requirements
ISO 15031-6	Road vehicles — Communication between vehicle and external equipment for emissions-related diagnostics — Part 6: Diagnostic Trouble Codes

2.2 SAE publications

SAE J1939-73	Recommended Practice for Serial Control and Communications Vehicle Network — Application Layer — Diagnostics

3 Terms and definitions

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

3.1
Integer type
A simple type with distinguished values which are the positive and the negative whole numbers, including zero. The range of type integer is not specified within this document.

3.2
Diagnostic trouble code
A numerical common identifier for a fault condition identified by the on-board diagnostic system.

3.3
Diagnostic Service
An information exchange initiated by a client in order to require diagnostic information from a server or/and to modify its behaviour for diagnostic purpose.

3.4
Client
The function that is part of the tester and that makes use of the diagnostic services. A tester normally makes use of other functions such as data base management, specific interpretation, human-machine interface.

3.5
Server
A function that is part of an electronic control unit and that provides the diagnostic services.

NOTE This international standard differentiates between the server (i.e. the function) and the electronic control unit so that this standard remains independent from the implementation.

3.6
Tester
A system that controls functions such as test, inspection, monitoring, or diagnosis of an on-vehicle electronic control unit and may be dedicated to a specific type of operators (e.g. a scan tool dedicated to garage mechanics or a test tool dedicated to assembly plant agents). The tester is also referenced as the client.

3.7
Diagnostic Data
Data that is located in the memory of an electronic control unit which may be inspected and/or possibly modified by the tester (diagnostic data includes analogue inputs and outputs, digital inputs and outputs, intermediate values and various status information).

NOTE Examples of diagnostic data: vehicle speed, throttle angle, mirror position, system status, etc..Three types of values are defined for diagnostic data: The current value: the value currently used by (or resulting from) the normal operation of the electronic control unit; A stored value: an internal copy of the current value made at specific moments (e.g. when a malfunction occurs or periodically); this copy is made under the control of the electronic control unit; A static value: e.g. VIN. The server is not obliged to keep internal copies of its data for diagnostic purposes, in which case the tester may only request the current value.

3.8
Diagnostic Session
The level of diagnostic functionality provided by the server.

NOTE Defining a repair shop or development testing session selects different server functionality (e.g. access to all memory locations may only be allowed in the development testing session).

3.9
Diagnostic Routine
A routine that is embedded in an electronic control unit and that may be started by a server upon a request from the client.

NOTE It could either run instead of a normal operating program, or could be enabled in this mode and executed with the normal operating program. In the first case, normal operation for the server is not possible. In the second case, multiple diagnostic routines may be enabled that run while all other parts of the electronic control unit are functioning normally.

3.10
Record
One or more diagnostic data elements that are referred to together by a single means of identification.

NOTE A snapshot including various input/output data and trouble codes is an example of a record.

3.11
Security
The security access method used in this document satisfies the requirements for tamper protection as specified in the ISO 15031-7 Road vehicles - Communication between vehicle and external equipment for emissions related diagnostics – Part 7: Data link security document.

3.12
Functional Unit
a set of functionally close or complementary diagnostic services.

3.13
O.S.I.
Open Systems Interconnection

3.14
ECU
Electronic Control Unit, also referenced as a server

NOTE Systems considered as Electronic Control Units: Anti-lock Braking System (ABS), Engine Management System....

3.15
Local server
a server that is connected to the same local network as the client and is part of the same address space as the client.

3.16
Local client
a client that is connected to the same local network as the server and is part of the same address space as the server.

3.17
Remote server
a remote server is a server that is not directly connected to the main diagnostic network. A remote server is identified by means of a remote address. Remote addresses represent an own address space that is independent from the addresses on the main network.

A remote server is reached via a local server on the main network. Each local server on the main network can act as a gate to one independent set of remote servers. A pair of addresses must therefore always identify a remote server: one local address that identifies the gate to the remote network and one remote address identifying the remote server itself.

3.18
Remote client
a remote client is a client that is not directly connected to the main diagnostic network. A remote client is identified by means of a remote address. Remote addresses represent an own address space that is independent from the addresses on the main network.

3.19
TA
Target Address

3.20
SA
Source Address

3.21
RA
Remote Address

3.22
TA_type
Target Address type

3.23
A_PCI
Application layer Protocol Control Information

3.24
SI
Service Identifier

3.25
NR_SI
Negative Response Service Identifier

4 Conventions

This international standard is guided by the conventions discussed in the O.S.I. Service Conventions (ISO/TR 8509) as they apply to the diagnostic services. These conventions specify the interactions between the service user and the service provider. Information is passed between the service user and the service provider by service primitives, which may convey parameters.

The distinction between service and protocol is summarized in the figure below.

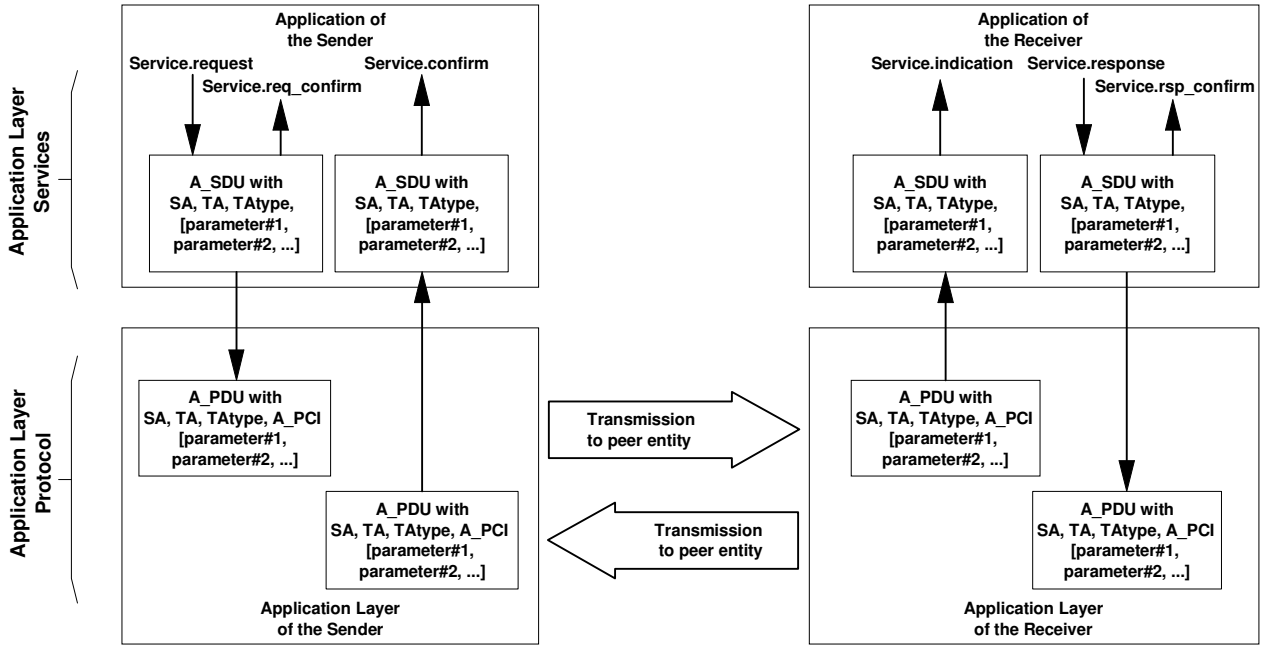


Figure 2 — The services and the protocol

ISO 14229 defines both, confirmed and unconfirmed services.

— The **confirmed services** use the six (6) service primitives request, req_confirm, indication, response, rsp_confirm and confirmation.

— The **unconfirmed services** use only the request, req_confirm and indication service primitives.

For all services defined in ISO 14229 the request and indication service primitives always has the same format and parameters. Consequently for all services the response and confirmation service primitives (except req_confirm and rsp_confirm) always have the same format and parameters. When the service primitives are defined in this International Standard, only the request and response service primitives are listed.

5 Application layer services

Application layer services are usually referred to as diagnostic services. The application layer services are used in client-server based systems to perform functions such as test, inspection, monitoring or diagnosis of on-board vehicle servers. The client, usually referred to as an External Test Equipment, uses the application layer services to request diagnostic functions to be performed in one or more servers. The server, usually a function that is part of an ECU, uses the application layer services to send response data, provided by the requested diagnostic service, back to the client. The client is usually an off-board tester, but can in some systems also be an on-board tester. The usage of application layer services is independent from the client being an off-board or on-board tester. It is possible to have more than one client in the same vehicle system.

5.1 General

The service access point of the diagnostics application layer provides a number of services that all have the same general structure. For each service, six (6) service primitives are specified:

— a **service request primitive**, used by the client function in the diagnostic tester application, to pass data about a requested diagnostic service to the diagnostics application layer;

— a **service request-confirmation primitive**, used by the client function in the diagnostic tester application, to indicate that the data passed in the service request primitive is completely transferred to the server;

— a **service indication primitive**, used by the diagnostics application layer, to pass data to the server function of the ECU diagnostic application;

— a **service response primitive**, used by the server function in the ECU diagnostic application, to pass response data provided by the requested diagnostic service to the diagnostics application layer;

— a **service response-confirmation primitive**, used by the server function in the ECU diagnostic application, to indicate that the data passed in the service response primitive is completely transferred to the client;

— a **service confirmation primitive** used by the diagnostics application layer to pass data to the client function in the diagnostic tester application.

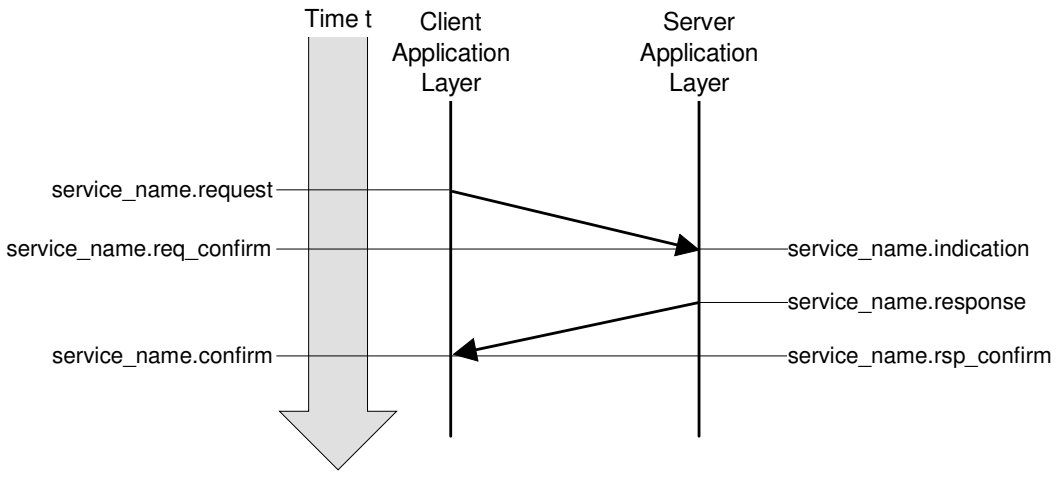


Figure 3 — Application layer service primitives - confirmed service

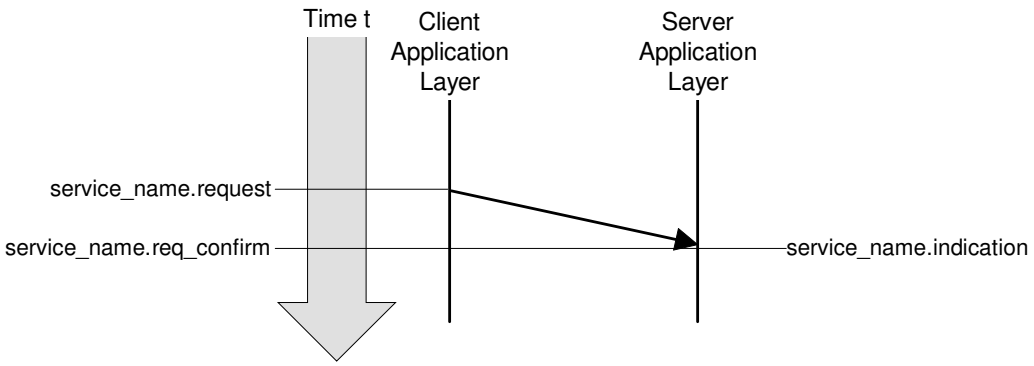


Figure 4 — Application layer service primitives - unconfirmed service

For a given service, the request primitive and the indication primitive always have the same service data unit. This International Standard will only list and specify the parameters of the service data unit belonging to each service request primitive. The user shall assume exactly the same parameters for each corresponding service indication primitive.

For a given service, the response primitive and the confirmation primitive always have the same service data unit. This International Standard will only list and specify the parameters of the service data unit belonging to each service response primitive. The user shall assume exactly the same parameters for each corresponding service confirmation primitive.

For each service response primitive (and corresponding service confirmation primitive) two different service data units (two sets of parameters) will be specified. One set of parameters shall be used in a positive service response primitive if the requested diagnostic service could be successfully performed by the server function in the ECU diagnostic application. Another service data unit shall be used in a negative service response primitive if the requested diagnostic service failed or could not be completed in time by the server function in the ECU diagnostic application.

For a given service, the request-confirmation primitive and the response-confirmation primitive always have the same service data unit. The purpose of these service primitives is to indicate the completion of an earlier request or response service primitive invokation. The service descriptions in in this International Standard will not make use of those service primitves, but the data link specific implementation documents might use those to define e.g. service execution reference points (e.g. the ECUReset service would invoke the reset when the response is completely transmitted to the client, which is indicated in the server via the service response-confirm primitive).

5.2 Format description of application layer services

Application layer services can have two different formats depending on how the vehicle diagnostic system is configured.

If the vehicle system is configured as one main (one logical) diagnostic network with all clients and servers directly connected, then the default (also called normal or standard) format of application layer services shall be used. This format is e.g. compatible with diagnostic system formats used on data links such as K- and L-lines. The default application layer services format is specified in clause 5.3.

The remote format of application layer services shall be used in vehicle systems implementing the concept of local servers and remote servers. The remote format has one additional address parameter called Remote address. The remote format is used to access servers that are not directly connected to the main diagnostic network in the vehicle. The remote format for application layer services is specified in clause 5.3.3.

5.3 Format description of standard service primitives

All application layer services have the same general format. Service primitives are written in the form:

service_name.type (
parameter A, parameter B, parameter C
[,parameter 1, ...]
)

Where:

— "service_name" is the name of the diagnostic service (e.g. DiagnosticSessionControl),

— "type" indicates the type of the service primitive (e.g. request),

— "parameter A, ..." is the A_SDU (Application layer Service Data Unit) as a list of values passed by the service primitive (addressing information),

— "parameter A, parameter B, parameter C" are mandatory parameters that shall be included in all service calls,

— "[,parameter 1, ...]" are parameters that depend on the specific service (e.g. parameter 1 can be the diagnosticSession for the DiagnosticSessionControl service). The brackets indicate that this part of the parameter list may be empty.

5.3.1 Service request and service indication primitves

For each application layer service, service request and service indication primitives are specified according to the following general format:

service_name.request (
SA,
TA,
TA_type
[,parameter 1, ...]
)

The request primitive is used by the client function in the diagnostic tester application, to initiate the service and pass data about the requested diagnostic service to the application layer.

service_name.indication (
SA,
TA,
TA_type
[,parameter 1, ...]
)

The indication primitive is used by the application layer, to indicate an internal event which is significant to the ECU diagnostic application and pass data about the requested diagnostic service to the server function of the ECU diagnostic application.

The request and indication primitive of a specific application layer service always have the same parameters and parameter values. This mean that the values of individual parameters shall not be changed by the communicating peer protocol entities of the application layer when the data is transmitted from the client to the server. The same values that are passed by the client function in the client application to the application layer in the service request call, shall be received by the server function of the diagnostic application from the service indication of the peer application layer.

5.3.2 Service response and service confirm primitives

For each confirmed application layer service, service response and service confirm primitives are specified according to the following general format:

```
service_name.response (
                        SA,
                        TA,
                        TA_type,
                        Result
                        [,parameter 1, ...]
                        )
```

The response primitive is used by the server function in the ECU diagnostic application, to initiate the service and pass response data provided by the requested diagnostic service to the application layer.

```
service_name.confirm  (
                        SA,
                        TA,
                        TA_type,
                        Result
                        [,parameter 1, ...]
                        )
```

The confirm primitive is used by the application layer to indicate an internal event which is significant to the client application and pass results of an associated previous service request to the client function in the diagnostic tester application. It does not necessarily indicate any activity at the remote peer interface, e.g if the requested service is not supported by the server or if the communication is broken.

The response and confirm primitive of a specific application layer service always have the same parameters and parameter values. This means that the values of individual parameters shall not be changed by the communicating peer protocol entities of the application layer when the data is transmitted from the server to the client. The same values that are passed by the server function of the ECU diagnostic application to the application layer in the service response call shall be received by the client function in the diagnostic tester application from the service confirmation of the peer application layer.

For each response and confirm primitive two different service data units (two sets of parameters) will be specified.

— A positive response and positive confirm primitive shall be used with the first service data unit if the requested diagnostic service could be successfully performed by the server function in the ECU.

— A negative response and confirm primitive shall be used with the second service data unit if the requested diagnostic service failed or could not be completed in time by the server function in the ECU.

5.3.3 Service request-confirm and service response-confirm primitives

For each application layer service, service request-confirm and service response-confirm primitives are specified according to the following general format:

```
service_name.req_confirm(
                        SA,
                        TA,
                        TA_type,
                        Result
                        )
```

The request-confirm primitive is used by the application layer to indicate an internal event, which is significant to the client application, and pass results of an associated previous service request to the client function in the diagnostic tester application.

service_name.rsp_confirm(
SA,
TA,
TA_type,
Result
)

The response-confirm primitive is used by the application layer to indicate an internal event, which is significant to the server application, and pass results of an associated previous service response to the server function in the ECU application.

5.4 Format description of remote service primitives

Diagnostic communication between a local client and a remote server can take place if the remote format of application layer services is used. All definitions made for the default format of application layer services shall be applicable also for the remote format of application layer services with the addition of one more addressing parameter.

Diagnostic communication can take place between a local client on the main network and one or more remote servers on a remote network. Communication can also take place between a remote client on a remote network and one or more local servers on the main network.

Diagnostic communication cannot take place between any combination of clients and servers on two different remote networks.

All remote format application layer services have the same general format. Service primitives are written in the form:

service_name.type (
parameter A, parameter B, parameter C,
parameter D
[,parameter 1, ...]
)

Where:

— "service_name" is the name of the diagnostic service (e.g. DiagnosticSessionControl),

— "type" indicates the type of the service primitive (e.g. request),

— "parameter A, ..." is the A_SDU (Application layer Service Data Unit) as a list of values passed by the service primitive (addressing information),

— "parameter A, parameter B, parameter C" are mandatory parameters that shall be included in all service calls,

— “parameter D” is an additional parameter that is only used in vehicles implementing the concept of remote servers (remote address),

— "[,parameter 1, ...]" are parameters that depend on the specific service (e.g. parameter 1 can be the diagnosticSession for the DiagnosticSessionControl service). The brackets indicate that this part of the parameter list may be empty.

5.4.1 Remote service request and service indication primitives

For each remote format application layer service, service request and service indication primitives are specified according to the following general format:

service_name.request (
SA,
TA,
TA_type
[,RA]
[,parameter 1, ...]
)

The request primitive is used by the local client function in the client application, to initiate the service and pass data about the requested diagnostic service to the application layer.

service_name.indication (
SA,
TA,
TA_type
[,RA]
[,parameter 1, ...]
)

The indication primitive is used by the remote application layer, to indicate an internal event, which is significant to the ECU diagnostic application and pass data about the requested diagnostic service to the remote server function of the ECU diagnostic application.

The request and indication primitive of a specific application layer service always have the same parameters and parameter values. This means that the values of individual parameters shall not be changed by the communicating peer protocol entities of the application layer when the data is transmitted from the client to the server. The same values that are passed by the client function in the diagnostic tester application to the application layer in the service request call shall be received by the server function of the ECU application from the service indication of the peer application layer.

NOTE For clarity the text assumes communication between a local client and one or more remote server. The protocol also supports communication between a remote client and one or more local servers using the same remote format application layer services.

5.4.2 Remote service response and service confirm primitives

For each remote format application layer service, service response and service confirm primitives are specified according to the following general format:

service_name.response (
SA,
TA,
TA_type,
[RA,]
Result
[,parameter 1, ...]
)

The response primitive is used by the remote server function in the ECU diagnostic application, to initiate the service and pass response data provided by the requested diagnostic service to the application layer.

service_name.confirm (
SA,
TA,

TA_type,
[RA,]
Result
[,parameter 1, ...]
)

The confirm primitive is used by the local application layer to indicate an internal event which is significant to the client application and pass results of an associated previous service request to the client function in the ECU application. It does not necessarily indicate any activity at the remote peer interface, e.g if the requested service is not supported by the server or if the communication is broken.

The response and confirm primitive of a specific application layer service always have the same parameters and parameter values. This means that the values of individual parameters shall not be changed by the communicating peer protocol entities of the application layer when the data is transmitted from the server to the client. The same values that are passed by the server function of the ECU diagnostic application to the application layer in the service response call shall be received by the client function in the diagnostic tester application from the service confirmation of the peer application layer.

For each response and confirm primitive two different service data units (two sets of parameters) will be specified.

— A positive response and positive confirm primitive shall be used with the first service data unit if the requested diagnostic service could be successfully performed by the server function in the ECU.

— A negative response and confirm primitive shall be used with the second service data unit if the requested diagnostic service failed or could not be completed in time by the server function in the ECU.

NOTE For clarity the text assumes communication between a local client and one or more remote server. The protocol also supports communication between a remote client and one or more local servers using the same remote format application layer services.

5.4.3 Remote service request-confirm and service response-confirm primitives

For each application layer service, service request-confirm and service response-confirm primitives are specified according to the following general format:

service_name.req_confirm(
SA,
TA,
TA_type,
[RA,]
Result
)

The request-confirm primitive is used by the client application layer to indicate an internal event, which is significant to the client application, and pass results of an associated previous service request to the client function in the ECU application.

service_name.rsp_confirm(
SA,
TA,
[RA,]
TA_type,
Result,
)

The response-confirm primitive is used by the server application layer to indicate an internal event, which is significant to the server application, and pass results of an associated previous service response to the server function in the ECU application.

5.5 Service data unit specification

5.5.1 Mandatory parameters

The application layer services contain three (3) mandatory parameters. The following parameter definitions are applicable to all application layer services specified in this International Standard (standard and remote format).

5.5.1.1 SA, Source Address

Type: 1 byte unsigned integer value

Range: 00-FF hex

Description:

The parameter SA shall be used to encode client and server identifiers and it shall be used to represent the physical location of a client or server.

For service requests (and service indications), SA represents the client identifier for the client function that has requested the diagnostic service. The client shall always be located in one diagnostic tester only. There shall be a strict, one to one, relation between client identifiers and source addresses. Each client identifier shall be encoded with one SA value. If more than one client is implemented in the same diagnostic tester, then each client shall have its own client identifier and corresponding SA value.

For service responses (and service confirmations), SA represents the physical location of the server that has performed the requested diagnostic service. A server may be implemented in one server only or be distributed and implemented in several ECUs. If a server is implemented in one ECU only, then it shall be encoded with one SA value only. If a server is distributed and implemented in several ECUs, then the server identifier shall be encoded with one SA value for each physical location of the server.

If a remote client or server is the original source for a message, then SA represents the local server that is the gate from the remote network to the main network.

NOTE The SA value in a response message will be the same as the TA value in the corresponding request message if physical addressing was used for the request message.

5.5.1.2 TA, Target Address

Type: 1 byte unsigned integer value

Range: 00-FF hex

Description:

The parameter TA shall be used to encode client and server identifiers.

Two different addressing methods, called:

— physical addressing, and

— functional addressing

are specified for diagnostics. Therefore, two independent sets of target addresses can be defined for a vehicle system (one for each addressing method).

Physical addressing shall always be a dedicated message to a server implemented in one ECU. When physical addressing is used, the communication is a point-to-point communication between the client and the server.

Functional addressing is used by the client if it does not know the physical address of the server that shall respond to a service request or if the server is implemented as a distributed server in several ECUs. When functional addressing is used, the communication is a broadcast communication from the client to a server implemented in one or more ECUs.

For service requests (and service indications), TA represents the server identifier for the server that shall perform the requested diagnostic service. If a remote server is being addressed, then TA represents the local server that is the gate from the main network to the remote network.

For service responses (and service confirmations), TA represents the client identifier for the client that originally requested the diagnostic service and shall receive the requested data. Service responses (and service confirmations) shall always use physical addressing. If a remote client is being addressed, then TA represents the local server that is the gate from the main network to the remote network.

NOTE The TA value of a response message will always be the same as the SA value of the corresponding request message.

5.5.1.3 TA_Type, Target Address type

Type: enumeration

Range: physical, functional

Description:

The parameter TA_type is an extension to the TA parameter. It is used to represent the addressing method chosen for a message transmission.

5.5.1.4 Result

Type: enumeration

Range: positive, negative

Description:

The parameter 'Result' is used by the response and confirm primitives to indicate if a message is a positive response/positive confirm message or a negative response/negative confirm message. The service specific parameters in the message are different depending on the value of the Result parameter.

5.5.2 Vehicle system requirements

The vehicle manufacturer shall ensure that each server in the system has a unique server identifier. The vehicle manufacturer shall also ensure that each client in the system has a unique client identifier.

All client and server identifiers for the main diagnostic network in a vehicle system shall be encoded into the same range of source addresses. This means that a client and a server shall not be represented by the same SA value in a given vehicle system.

The physical target address for a server shall always be the same as the source address for the server.

Remote server identifiers can be assigned independently from client and server identifiers on the main network.

In general only the server(s) addressed shall respond to the client request message.

5.5.3 Optional parameters

5.5.3.1 RA, Remote Address

Type: 1 byte unsigned integer value

Range: 00-FF hex

Description:

RA is used to extend the available address range to encode client and server identifiers. RA shall only be used in vehicles that implement the concept of local servers and remote servers. Remote addresses represents its own address range and are independent from the addresses on the main network.

The parameter RA shall be used to encode remote client and server identifiers. RA can represent either a remote target address or a remote source address depending on the direction of the message carrying the RA.

For service requests (and service indications) sent by a client on the main network, RA represents the remote server identifier (remote target address) for the server that shall perform the requested diagnostic service.

RA can be used both as a physical and a functional address. For each value of RA, the system builder shall specify if that value represents a physical or functional address.

NOTE There is no special parameter that represents physical or functional remote addresses in the way TA_type specifies the addressing method for TA. Physical and functional remote addresses share the same 1 byte range of values and the meaning of each value shall be defined by the system builder.

For service responses (and service confirmations) sent by a remote server, RA represents the physical location (remote source address) of the remote server that has performed the requested diagnostic service.

A remote server may be implemented in one ECU only or be distributed and implemented in several ECUs. If a remote server is implemented in one ECU only, then it shall be encoded with one RA value only. If a remote server is distributed and implemented in several ECUs, then the remote server identifier shall be encoded with one RA value for each physical location of the remote server.

For service requests (and service indications) sent by a remote client, RA represents the remote server identifier (remote source address) for the client function that has requested the diagnostic service.

For service responses (and service confirmations) sent by a local server, RA represents the remote client identifier (remote target address) for the client that originally requested the diagnostic service and shall receive the requested data.

5.5.3.2 Remote server example with remote network

In some systems the remote server is connected to a remote network separated from the main diagnostic network by a gateway. The following is an example showing how the parameters SA, TA and RA shall be used for proper communication between a local client on the main network and a remote server via a gateway. In the example it is assumed that the same type of addressing is used on the remote network as on the main network.

The External test equipment is connected to the main network and has client identifier 241 (F1 hex). The Gate is connected to both the main network and the remote network. On the main network the Gate has client identifier 200 (C8 hex). On the remote network the Gate has client identifier 10 (0A hex). The Remote server is connected to the remote network and has client identifier 62 (3E hex). The configuration is described in the figure below.

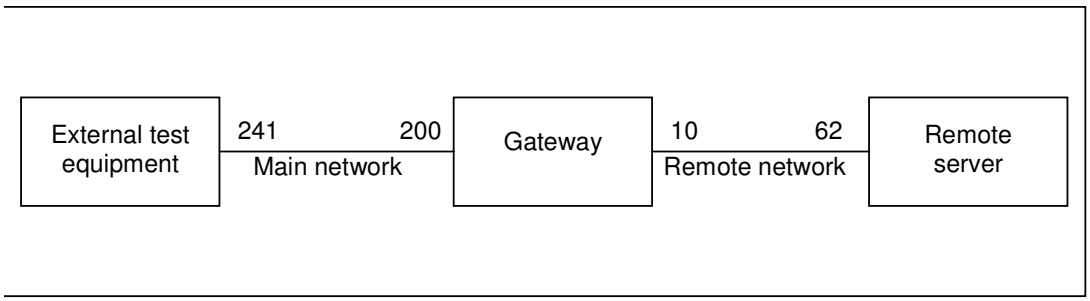


Figure 5 — Remote server system example 1

The External test equipment sends a remote diagnostic request message with

— SA = 241 (F1 hex),

— TA = 200 (C8 hex), and

— RA = 62 (3E hex).

The Gate receives the message and sends it out on the remote network with

— SA = 10 (0A hex),

— TA = 62 (3E hex), and

— RA = 241 (F1 hex).

The Remote server receives the message.

The Remote server send back a remote diagnostic response message with

— SA = 62 (3E hex),

— TA = 10 (0A hex) and

— RA = 241 (F1 hex).

The Gate receives the message and sends it out on the main network with

— SA = 200 (C8 hex),

— TA = 241 (F1 hex) and

— RA = 62 (3E hex).

The External test equipment receives the message.

5.5.3.3 Remote server example without remote network

In some systems the remote server is a functional part of a server belonging to the main network. The server has been given a remote server identifier in order to extend the available address range to encode client and server identifiers. In such systems the Remote server is logically separated from the main network even if the ECU, that the Remote server is part, is connected to the main diagnostic network. To get a working system, the server must also have a gateway function that is part of the main diagnostic network and can serve as a gate to the Remote server. The following is an example showing how the parameters SA, TA and RA shall be

used for proper communication between a local client on the main network and a remote server via a gateway.

The External test equipment is connected to the main network and has client identifier 241 (F1 hex). The Gate is connected to the same main network. The Gate has client identifier 200 (C8 hex). The Remote server has client identifier 62 (3E hex). The configuration is described in the figure below.

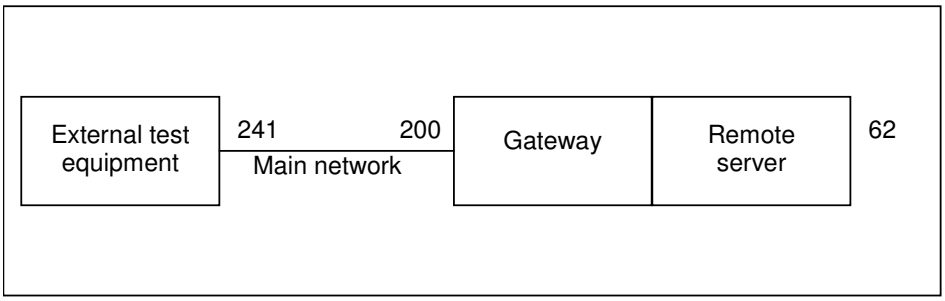


Figure 6 — Remote server system example 2

The External test equipment sends a remote diagnostic request message with

— SA = 241 (F1 hex),

— TA = 200 (C8 hex), and

— RA = 62 (3E hex).

The Gate receives the message and passes it over to the Remote server function. The Remote server receives the message.

The Remote server sends back a remote diagnostic response message by passing it to the gateway function. The Gate receives the message and sends it out on the main network with

— SA = 200 (C8 hex),

— TA = 241 (F1 hex), and

— RA = 62 (3E hex).

The External test equipment receives the message.

5.5.3.4 Remote client example with remote network

In some systems the client is connected to a remote network separated from the main diagnostic network by a gateway. The following is an example showing how the parameters SA, TA and RA shall be used for proper communication between a remote client on a remote network and a local server on the main network via a gateway. In the example it is assumed that the same type of addressing is used on the remote network as on the main network.

The External test equipment is connected to the remote network and has client identifier 242 (F2 hex). The Gate is connected to both the main network and the remote network. On the main network the Gate has client identifier 200 (C8 hex). On the remote network the Gate has client identifier 10 (0A hex). The local server is connected to the main network and has client identifier 18 (12 hex). The configuration is described in the figure below.

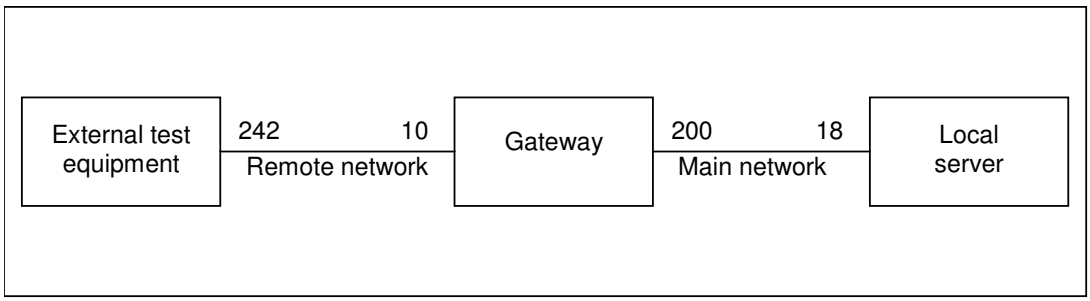


Figure 7 — Remote client example

The External test equipment sends a remote diagnostic request message with

— SA = 242 (F1 hex),

— TA = 10 (0A hex), and

— RA = 18 (12 hex).

The Gate receives the message and sends it out on the main network with SA = 200 dec, TA = 18 dec and RA = 242 dec. The local server receives the message.

The local server sends back a remote diagnostic response message with

— SA = 18 (12 hex),

— TA = 200 (C8 hex), and

— RA = 242 (F1 hex).

The Gate receives the message and sends it out on the remote network with

— SA = 10 (0A hex),

— TA = 242 (F1 hex), and

— RA = 18 (12 hex).

The External test equipment receives the message.

6 Application layer protocol

6.1 General

The application layer protocol shall always be a confirmed message transmission, meaning that for each service request sent from the client, there shall be one or more corresponding responses sent from the server.

The only exception from this rule shall be a few cases when e.g. functional addressing is used or the request/indication specifies that no response/confirmation shall be generated. In order not to burden the system with many unnecessary messages, there are a few cases when a negative response messages shall not be sent even if the server failed to complete the requested diagnostic service.

The application layer protocol shall be handled in parallel with the session layer protocol. This mean that even if the client is waiting for a response to a previous request, it shall maintain proper session layer timing (e.g. sending a TesterPresent request if that is needed to keep a diagnostic session going in other servers. The implementation depends on the data link layer used).

6.2 Protocol data unit specification

The A_PDU (Application layer Protocol Data Unit) is directly constructed from the A_SDU (Application layer Service Data Unit) and the layer specific control information A_PCI (Application layer Protocol Control Information). The A_PDU shall have the following general format:

A_PDU (
SA,
TA,
TA_type,
[RA,]
A_Data = A_PCI + [parameter 1, ...]
)

Where:

— "SA, TA, TA_type, RA" are the same parameters as used in the A_SDU,

— "A_Data" is a string of byte data defined for each individual application layer service. The A_Data string shall start with the A_PCI followed by all service specific parameters from the A_SDU as specified for each service. The brackets indicate that this part of the parameter list may be empty.

6.3 Application protocol control information

The A_PCI shall have two alternative formats depending on which type of service primitive that has been called and the value of the Result parameter. For all service requests and for service responses/service confirmations with Result = positive, the following definition shall apply:

A_PCI (
SI
)

Where:

— "SI" is the parameter Service identifier.

For service responses/service confirmations with Result = negative, the following definition shall apply:

A_PCI (
NR_SI,

SI
)

Where:

— "NR_SI" is the special parameter identifying negative service responses/confirmations,

— "SI" is the parameter Service identifier.

NOTE For the transmission of periodic messages utilizing response message type #2 as defined in the service ReadDataByPeriodicIdentifier (2A hex, see section 9.4) no A_PCI is present in the application layer protocol data unit (A_PDU).

6.3.1 SI, Service Identifier

Type: 1 byte unsigned integer value

Range: 00-FF hex according to definitions in the table below.

Table 1 — Service identifier (SI) values

Service identifier (hex value)	Service type (bit 6)	Where defined
00 – 0F	OBD service requests	ISO 15031-5
10 – 3E	ISO 14229-1 service requests	ISO 14229-1
3F	Not applicable	Reserved by document
40 – 4F	OBD service responses	ISO 15031-5
50 – 7E	ISO 14229-1 positive service responses	ISO 14229-1
7F	Negative response service identifier	ISO 14229-1
80	Not applicable	Reserved by ISO 14229-1
81 – 82	Not applicable	Reserved by ISO 14230
83 – 87	ISO 14229-1 service requests	ISO 14229-1
88 – 9F	Service requests	Reserved for future exp. As needed
A0 – B9	Service requests	Defined by vehicle manufacturer
BA – BE	Service requests	Defined by system supplier
BF	Not applicable	Reserved by document
C0	Not applicable	Reserved by ISO 14229-1
C1 – C2	Not applicable	Reserved by ISO 14230
C3 – C7	ISO 14229-1 positive service responses	ISO 14229-1
C8 – DF	Positive service responses	Reserved for future exp. as needed
E0 – F9	Positive service responses	Defined by vehicle manufacturer
FA – FE	Positive service responses	Defined by system supplier
FF	Not applicable	Reserved by document

NOTE There is a one-to-one correspondence between service identifiers for request messages and service identifiers for positive response messages, with bit 6 of the SI hex value indicating the service type. All request messages have SI bit 6 = 0. All positive response messages have SI bit 6 = 1, except response message type #2 of the ReadDataByPeriodicIdentifier (2A hex, see section 9.4) service.

Description:

The SI shall be used to encode the specific service that has been called in the service primitive. Each request service shall be assigned a unique SI value. Each positive response service shall be assigned a corresponding unique SI value.

The service identifier is used to represent the service in the A_Data data string that is passed from the application layer to lower layers (and returned from lower layers).

6.3.2 NR_SI, Negative response service identifier

Type: 1 byte unsigned integer value

Fixed value: 7F hex

Description:

The parameter NR_SI is a special parameter identifying negative service responses/confirmations. It shall be part of the A_PCI for negative response/confirm messages.

NOTE The NR_SI value is co-ordinated with the SI values. The NR_SI value is not used as a SI value in order to make A_Data coding and decoding easier.

6.4 Negative response/confirmation service primitive

The following sections specify the behaviour of the server when executing a service.

6.4.1 Physically addressed request message and server response behaviour

The following communication schemes are possible:

a) The server shall send a **positive** response message if the requested diagnostic service (ResponseRequired = **YES**) could be correctly interpreted.

b) The server shall send a **negative** response message if the requested diagnostic service (ResponseRequired = **YES**) **failed** or could **not be completed in time**.

c) The server shall **NOT send** a response message if the requested diagnostic service (ResponseRequired = **NO**) could be correctly interpreted.

d) The server shall send a **negative** response message if the requested diagnostic service (ResponseRequired = **NO**) **failed** or could **not be completed in time**.

Table 2 — Physically addressed request message and server response behaviour

Client request message		Server capabilityRequest			Server response		Comments to server response behaviour
Addressing scheme	**Sub-Function (RespReq'd)**	**ServiceID supported**	**Sub-F. supported**	**Data bytes supported**	**Message**	**Negative: NRC/section**	
physical	YES	YES	YES	YES	Positive	---	Server sends positive response
physical	YES	**NO**	---	---	**Negative**	NRC=11 hex	Negative response with **NRC** 11 hex
physical	YES	YES	**NO**	---	**Negative**	NRC=12 hex	Negative response with **NRC** 12 hex
physical	YES	YES	YES	**NO**	**Negative**	Section x.x.4	Negative response with **NRC** defined by service
physical	**NO**	YES	YES	YES	**NO**	---	Server does **NOT** send a response
physical	**NO**	**NO**	---	---	**Negative**	NRC=11 hex	Negative response with **NRC** 11 hex
physical	**NO**	YES	**NO**	---	**Negative**	NRC=12 hex	Negative response with **NRC** 12 hex
physical	**NO**	YES	YES	**NO**	**Negative**	Section x.x.4	Negative response with **NRC** defined by service

6.4.2 Functionally addressed request message and server response behaviour

The following communication schemes are possible:

a) The server shall send a **positive** response message if the requested diagnostic service (ResponseRequired = **YES**) could be correctly interpreted.

b) The server shall NOT send a response message if the requested diagnostic service (ResponseRequired = YES) is not supported, or if the service is supported but the sub-function is not supported. If the requested diagnostic service failed due to any other reason or could not be completed in time, then the server shall send a negative response message.

c) The server shall **NOT send** a response message if the requested diagnostic service (ResponseRequired = **NO**) could be correctly interpreted.

d) The server shall **NOT** send a **negative** response message if the requested diagnostic service (ResponseRequired = **NO**) is not supported, or if the service is supported but the sub-function is not supported. The server shall send a negative resonse message if the requested diagnostic service **failed** due to any other reason or could **not be completed in time** (negative response code different from serviceNotSupported or subFunctionNotSupported).

Table 3 — Functionally addressed request message and server response behaviour

Client request message		Server capabilityRequest			Server response		Comments to server response behaviour
Addressing scheme	**Sub-Function (RespReq'd)**	**ServiceID supported**	**Sub-F. supported**	**Data bytes supported**	**Message**	**Negative: NRC/section**	
functional	YES	YES	YES	YES	Positive	---	Server sends positive response
Functional	YES	**NO**	---	---	**NO**	---	Server does **NOT** send a response
Functional	YES	YES	**NO**	---	**NO**	---	Server does **NOT** send a response
Functional	YES	YES	YES	**NO**	**Negative**	Section x.x.4	Negative response with **NRC** defined by service
Functional	**NO**	YES	YES	YES	**NO**	---	Server does **NOT** send a response
Functional	**NO**	**NO**	---	---	**NO**	---	Server does **NOT** send a response
Functional	**NO**	YES	**NO**	---	**NO**	---	Server does **NOT** send a response
functional	**NO**	YES	YES	**NO**	**Negative**	Section x.x.4	Negative response with **NRC** defined by service

6.4.3 Pseudo code example of server response behaviour

The following is a server pseudo code example to describe the logical steps a server shall perform when receiving a request from the client.

Table 4 — Pseudo code example of server response behaviour

if (A_PDU.A_Data.A_PCI.SI == notSupported) **then** (NRC == ServiceNotSupported)		
if (A_PDU.A_Data.Parameter1 == notSupported) **then** (NRC == SubFunctionNotSupported)		
if ((responseRequiredIndicationBit == 1) **AND** (response is positive))		
then	**else**	
suppress response	**If** (	
	(	
	NRC == ServiceNotSupported /* 11 hex */	
	OR	
	NRC == SubFunctionNotSupported /* 12 hex */	
	)	
	AND	
	A_PDU.TA_type == functional /* functional request */	
	)	
	then	**else**
	suppress response message	send response message

Legend:

A_PDU.TA_type	Addressing scheme of request message sent by the client
A_PDU.A_Data.A_PCI.SI	Request Service Iidentifier
A_PDU.A_Data.Parameter1	sub-function parameter of a service request message
ServiceNotSuppported	service is not suppported
SubFunction	sub-function parameter of a service request message
SubFunctionNotSupported	sub-function is not supported
responseRequiredIndicationBit	response required indication bit is bit 7 of the sub-function parameter
addressing	Addressing scheme of request message sent by the client
functional	Functionally addressed request messge sent by the client

6.4.4 Negative response/negative confirmation message

Each diagnostic service has a negative response/negative confirmation message specified with message A_Data bytes according to the table below. The first A_Data byte (A_PCI.NR_SI) is always the specific negative response service identifier. The second A_Data byte (A_PCI.SI) shall be a copy of the service identifier value from the service request/indication message that the negative response message corresponds to.

Table 5 — Negative response A_PDU

A_PDU parameter	Parameter Name	Cvt	Hex Value	Mnemonic
SA	Source Address	M	xx	SA
TA	Target Address	M	xx	TA
TAtype	Target Address type	M	xx	TAT
RA	Remote Address (optional)	C	xx	RA
A_Data.A_PCI.NR_SI	Negative Response Service Id	M	7F	SIDNR
A_Data.A_PCI. SI	<Service Name> Request Service Id	M	xx	SIDRQ
A_Data.Parameter 1	responseCode	M	xx	NRC_
M (Mandatory):	In case the negative response A_PDU is issued then those A_PDU parameters shall be present.			
C (Conditional):	The RA (Remote Address) PDU parameter is only present in case of remote addressing.			

NOTE A_Data represents the message data bytes of the negative response message.

The parameter responseCode is used in the negative response message to indicate why the diagnostic service failed or could not be completed in time. Values are defined in annex A.1.

7 Service description conventions

This section defines how each diagnostic service is described in this specification. It defines the general service description format of each diagnostic service.

7.1.1 Service description

This clause gives a brief outline of the functionality of the service. Each diagnostic service specification starts with a description of the actions performed by the client and the server(s), which are specific to each service. The description of each service includes a table, which lists the parameters of its primitives: request/indication, response/confirmation for positive or negative result. All have the same structure:

For a given request/indication and response/confirmation A_PDU definition the presence of each parameter is described by one of the following convention (Cvt) values:

Table 6 — A_PDU parameter conventions

Type	Name	Description
M	Mandatory	The parameter has to be present in the A_PDU.
C	Conditional	The parameter can be present in the A_PDU, based on certain criteria (e.g. sub-function/parameters within the A_PDU).
S	Selection	Indicates, that the parameter is mandatory (unless otherwise specified) and is a selection from a parameter list.
U	User option	The parameter may or may not be present, depending on dynamic usage by the user.
NOTE The "<Service Name> Request Service Id" marked as 'M' (Mandatory), shall not imply that this service has to be supported by the server. The 'M' only indicates the mandatory presence of this parameter in the request A_PDU in case the server supports the service.		

7.1.2 Request message definition

This section includes one or multiple tables, which define the A_PDU (Application layer protocol data unit, see section 6) parameters for the service request/indication. There might be a separate table for each sub-function parameter ($Level) in case the request messages of the different sub-function parameters ($Level) differ in the structure of the A_Data parameters and cannot be specified clearly in one table.

Table 7 — Request A_PDU definition with sub-function

A_PDU parameter	Parameter Name	Cvt	Hex Value	Mnemonic
SA	Source Address	M	xx	SA
TA	Target Address	M	xx	TA
TAtype	Target Address type	M	xx	TAT
RA	Remote Address	C	xx	RA
A_Data.A_PCI.SI	<Service Name> Request Service Id	M	xx	SIDRQ
A_Data. Parameter 1	sub-function = [parameter]	S	 xx	LEV_ PARAM
Parameter 2 : Parameter k	data-parameter#1 : data-parameter#k-1	U : U	xx : xx	DP_…#1 : DP_…#k-1
C: The RA (Remote Address) PDU parameter is only present in case of remote addressing.				

Table 8 — Request A_PDU definition without sub-function

A_PDU parameter	Parameter Name	Cvt	Hex Value	Mnemonic
SA	Source Address	M	xx	SA
TA	Target Address	M	xx	TA
TAtype	Target Address type	M	xx	TAT
RA	Remote Address	C	xx	RA
A_Data.A_PCI.SI	<Service Name> Request Service Id	M	xx	SIDRQ
A_Data.. Parameter 1 : Parameter k	 data-parameter#1 : data-parameter#k	 U : U	 xx : xx	 DP_…#1 : DP_…#k
C: The RA (Remote Address) PDU parameter is only present in case of remote addressing.				

In all requests/indications the addressing information TA, SA, and TAtype is mandatory. The addressing information RA is optionally to be present.

NOTE The addressing information is shown in the table above for definition purpose. Further service request/indication definitions only specify the A_Data A_PDU parameter, because the A_Data A_PDU parameter represents the message data bytes of the service request/indication.

7.1.2.1 Request message sub-function parameter $Level (LEV_) definition

This section defines the sub-function $levels (LEV_) parameter(s) defined for the request/indication of the service <Service Name>.

This section does not contain any definition in case the described service does not use a sub-function parameter value and does not utilize the responseRequiredIndicationBit (this implicitly indicates that a response is required).

The sub-function parameter byte is divided into two parts (on bit-level) as defined in the table below.

Table 9 — Sub-function parameter structure

Bit position	Description
7	**responseRequiredIndicationBit** This bit indicates if a response is required from the server. '0' = yes, a response is required. '1' = no, a response is not required; the server(s) being addressed shall not send a positive response message. Independent of the responseRequiredIndicationBit, negative response messages are sent by the server(s) according to the restrictions specified in "" (1.1). ISO 14229-1 does not define the values of the sub-function parameter for the service where the responseRequiredIndicationBit is present. The applicable values for this parameter are defined in the implementation specifications of ISO 14229-1.
6-0	**sub-function parameter value** The bits 0-6 of the sub-function parameter contain the sub-function parameter value of the service (00 - 7F hex). Each service utilizing the sub-function parameter byte, but only supporting the responseRequiredIndicationBit has to support the zeroSubFunction sub-function parameter value (00 hex).

The sub-function parameter value is a 7 bit value (bits 6-0 of the sub-function parameter byte) that can have multiple values to further specify the service behaviour.

Each service only supporting the responseRequiredIndicationBit has to support the zeroSubFunction (00 hex).

Services supporting sub-function parameter values in addition to the responseRequiredIndicationBit shall support the sub-function parameter values as defined in the sub-function parameter value table.

Each service contains a table that defines values for the sub-function parameter values, taking only into account the bits 0-6.

Table 10 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
xx	**sub-function#1** description of sub-function parameter#1	M/U	SUBFUNC1
:	:	:	:
xx	**sub-function#m** description of sub-function parameter#m	M/U	SUBFUNCm

The convention (Cvt) column in the table above shall be interpreted as follows:

Table 11 — Sub-function parameter conventions

Type	Name	Description
M	Mandatory	The sub-function parameter has to be supported by the server in case the service is supported.
U	User option	The sub-function parameter may or may not be supported by the server, depending on the usage of the service.

The complete sub-function parameter byte value is calculated based on the value of the responseRequiredIndicationBit and the sub-function parameter value chosen.

Table 12 — Calculation of the sub-function byte value

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
responseRequired IndicationBit	Sub-function parameter value as specified in the sub-function parameter value table of the service						
resulting sub-function parameter byte value (bit 7 - 0)							

7.1.2.2 Request message data parameter definition

This section defines the data-parameter(s) $DataParam (DP_) for the request/indication of the service <Service Name>. This section does not contain any definition in case the described service does not use any data parameter. The data parameter portion can contain multiple bytes. This section provides a generic description of each data parameter, detailed definitions can be found in the annex of this document. The annex also specifies whether a data parameter shall be supported or is user optional to be supported in case the server supports the service.

Table 13 — Request message data parameter definition

Definition
data-parameter#1 description of data-parameter#1
:
data-parameter#n description of data-parameter#n

7.1.3 Positive response message definition

This section includes one or multiple tables that define the A_PDU parameters for the service response/confirmation (see section 6 for a detailed description of the application layer protocol data unit A_PDU). There might be a separate table for each sub-function parameter $Level when the response messages of the different sub-function parameters $Level differ in the structure of the A_Data parameters.

NOTE The positive response message of a diagnostic service (if required) shall be sent after the execution of the diagnostic service. In case a diagnostic service requires a different handling (e.g. ECUReset service) then the appropriate description when to sent the positive response message can be found in the service description of the diagnostic service.

Table 14 — Positive response A_PDU

A_PDU parameter	Parameter Name	Cvt	Hex Value	Mnemonic
SA	Source Address	M	xx	SA
TA	Target Address	M	xx	TA
TAtype	Target Address type	M	xx	TAT
RA	Remote Address	C	xx	RA
A_Data.A_PCI.SI	<Service Name> Response Service Id	S	xx	SIDPR
A_Data.Parameter 1 : A_Data.Parameter n	data-parameter#1 : data-parameter#n	U	xx : xx	DP_…#1 : DP_…#n
C: The RA (Remote Address) PDU parameter is only present in case of remote addressing.				

In all responses/confirmations the addressing information TA, SA, and TAtype is mandatory. The addressing information RA is optionally to be present.

NOTE The addressing information is shown in the table above for definition purpose. Further service request/indication definitions only specify the A_Data A_PDU parameter, because the A_Data A_PDU parameter represents the message data bytes of the service response/confirmation.

7.1.3.1 Positive response message data parameter definition

This section defines the data-parameter(s) for the response/confirmation of the service <Service Name>. This section does not contain any definition in case the described service does not use any data parameter. The data parameter portion can contain multiple bytes. This section provides a generic description of each data parameter, detailed definitions can be found in the annex of this document. The annex also specifies whether a data parameter shall be supported or is user optional to be supported in case the server supports the service.

Table 15 — Response data parameter definition

Definition
data-parameter#1 description of data-parameter#1. In case the request supports a sub-function parameter then this parameter is an echo of the sub-function parameter from the request message.
:
data-parameter#m description of data-parameter#m

7.1.4 Supported negative response codes (NRC_)

This section defines the negative response codes that shall be implemented for this service. The circumstances under which each response code would occur are documented in a table as given below. The definition of the negative response message can be found in section 1.1. The server shall use the negative response A_PDU for the indication of an identified error condition.

There are certain negative response codes, which are not related to the behaviour of the service and have to be supported in general. Details can be found in annex A.1.

Table 16 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
xx	**NegativeResponseCode#1** 1. condition#1 : m. condition #m	M	NRC_
:	:	U	NRC_
xx	**NegativeResponseCode#n** 1. condition#1 : k. condition #k	U	NRC_

The convention (Cvt) column in the table above shall be interpreted as follows:

Table 17 — Sub-function parameter conventions

Type	Name	Description
M	Mandatory	The negative response code has to be supported by the server in case the service is supported.
U	User option	The negative response code may or may not be supported by the server, depending on the usage of the service.

7.1.5 Message flow examples

This section contains message flow examples for the service <Service Name>. All examples are shown on a message level (without addressing information).

Table 18 — Request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1 (A_PCI)	<Service Name> request Service Id	xx	SIDRQ
#2	sub-function/data-parameter#1	xx	LEV_/DP_
:	:	xx	DP_
#n	data-parameter#m	xx	DP_

Table 19 — Positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1 (A_PCI)	<Service Name> response Service Id	xx	SIDPR
#2	data-parameter#1	xx	DP_
:	:	:	:
#n	data-parameter#n-1	xx	DP_

There might be multiple examples if applicable to the service <Service Name> (e.g. one for each sub-function parameter $Level).

The following table shows a message flow example for a negative response message.

Table 20 — Negative response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1 (A_PCI.NR_SI)	Negative Response Service Id	7F	SIDRSIDNRQ
#2 (A_PCI.SI)	<Service Name> request Service Id	xx	SIDRQ
#3	responseCode	xx	NRC_

8 Diagnostic and Communication Management functional unit

Table 21 — Diagnostic and Communication Management functional unit

Service	Description
DiagnosticSessionControl	The client requests to control a diagnostic session with a server(s).
EcuReset	The client forces the server(s) to perform a reset.
SecurityAccess	The client requests to unlock a secured server(s).
CommunicationControl	The client requests the server to control its communication.
TesterPresent	The client indicates to the server(s) that it is still present.
AccessTimingParameter	The client uses this service to read/modify the timing parameters for an active communication.
SecuredDataTransmission	The client uses this service to perform data transmission with an extended data link security.
ControlDTCSetting	The client controls the setting of DTCs in the server.
ResponseOnEvent	The client requests to start an event mechanism in the server.
LinkControl	The client requests control of the communication baudrate.

8.1 DiagnosticSessionControl (10 hex) service

The DiagnosticSessionControl service is used to enable different diagnostic sessions in the server(s).

8.1.1 Service description

A diagnostic session enables a specific set of diagnostic services and/or functionality in the server(s). It furthermore can enable a data link layer dependent set of timing parameters applicable for the started session. This service provides the capability that the server(s) can report data link layer specific parameter values valid for the enabled diagnostic session (e.g. timing parameter values). The data link layer specific implementation document defines the structure and content of the optional parameter record contained in the response message of this service. The user of this International Standard shall define the exact set of services and/or functionality enabled in each diagnostic session (superset of functionality that is available in the defaultSession).

There shall always be exactly one diagnostic session active in a server. A server shall always start the default diagnostic session when powered up. If no other diagnostic session is started, then the default diagnostic session shall be running as long as the server is powered.

A server shall be capable of providing diagnostic functionality under normal operating conditions and in other operation conditions defined by the vehicle manufacturer, e.g. limp home operation condition.

If the client has requested a diagnostic session, which is already running, then the server shall send a positive response message and behave as shown in the figure below that describes the server internal behaviour when transitioning between sessions.

Whenever the client requests a new diagnostic session, the server shall first send a DiagnosticSessionControl positive response message before the new session becomes active in the server. If the server is not able to start the requested new diagnostic session, then it shall respond with a DiagnosticSessionControl negative response message and the current session shall continue (see diagnosticSession parameter definitions for further information on how the server and client shall behave). There shall be only one session active at a time. A diagnostic session enables a specific set of diagnostic services and functions, which shall be defined by the vehicle manufacturer. The set of diagnostic services and diagnostic functionality in a non-default diagnostic session is a superset of the functionality provided in the defaultSession, which means that the diagnostic functionality of the defaultSession is also available when switching to any non-default diagnostic

session. A session can enable vehicle manufacturer specific services and functions, which are not part of this document.

To start a new diagnostic session a server may request that certain conditions be fulfilled. All such conditions are user defined. An example of such a condition is:

— The server may only allow a client with a certain client identifier (client diagnostic address) to start a specific new diagnostic session (e.g. a server may require that only a client having the client identifier F4 hex may start the extendedDiagnosticSession).

— In some systems it is desirable to change communication-timing parameters when a new diagnostic session is started. The DiagnosticSessionControl service entity can use the appropriate service primitives to change the timing parameters as specified for the underlying layers to change communication timing in the local node and potentially in the nodes the client wants to communicate with.

The following figure provides an overview about the diagnostic session transition and what the server shall do when it transitions to another session.

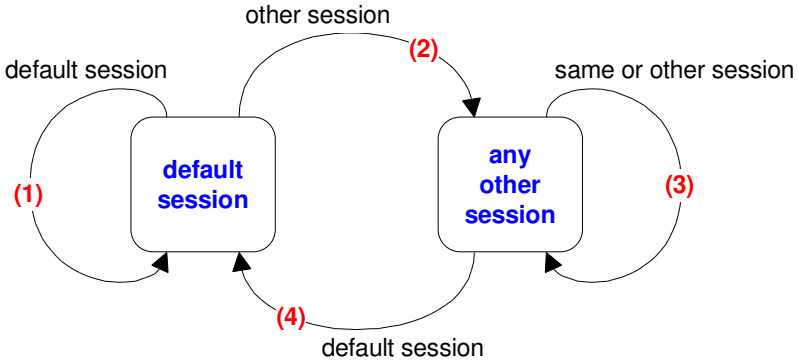


Figure 8 — Server diagnostic session state diagram

Diagnostic session transition description:

1) When the server is in the defaultSession and the client requests to start the defaultSession then the server shall re-initialize the defaultSession completely, which means that e.g. each event that has been configured in the server via the ResponseOnEvent (86 hex) service shall be reset. The server shall reset all activated/initiated/changed settings/controls during the activated session. This does not include long term changes programmed into non-volatile memory.

2) When the server transitions from the defaultSession to any other session than the defaultSession then the server shall only reset the events that have been configured in the server via the ResponseOnEvent (86 hex) service during the defaultSession.

3) When the server transitions from any diagnostic session other than the defaultSession to another session other than the defaultSession (including the currently active diagnostic session) then the server shall (re-) initialize the diagnostic session, which means that each event that has been configured in the server via the ResponseOnEvent (86 hex) service shall be reset and that security shall be enabled. Any configured periodic scheduler shall remain active when transitioning from one non-defaultSession to another or the same non-DefaultSession. The states of the CommunicationControl and ControlDTCSetting services shall not be affected, which e.g. means that normal communication shall remain disabled when it is disabled at the point in time the session is switched.

4) When the server transitions from any diagnostic session other than the default session to the defaultSession then the server shall reset each event that has been configured in the server via the ResponseOnEvent (86 hex) service and security shall be enabled. Any configured periodic

scheduler shall be disabled. Furthermore the states of the CommunicationControl and ControlDTCSetting services shall be reset, which e.g. means that normal communication shall be re-enabled when it was disabled at the point in time the session is switched to the defaultSession. The server shall reset all activated/initiated/changed settings/controls during the activated session. This does not include long term changes programmed into non-volatile memory.

The following table shows the services, which are allowed during the defaultSession and the non-defaultSession (timed services). Any non-defaultSession is tied to a diagnostic session timer that has to be kept active by the client.

Table 22: Services allowed during default and non-default diagnostic session

Service	defaultSession	non-defaultSession
1) It is implementation specific whether the ResponseOnEvent service is also allowed during the defaultSession. 2) Secured recordDataIdentifiers require a SecurityAccess service and therefore a non-default diagnostic session. 3) Secured memory areas require a SecurityAccess service and therefore a non-default diagnostic session. 4) A recordDataIdentifier can be defined dynamically in the default and non-default diagnostic session. 5) Secured routines require a SecurittAccess service and therefore a non-default diagnostic session. A routine that requires to be stopped actively by the client also requires a non-default session.		
DiagnosticSessionControl - 10 hex	x	x
ECUReset - 11 hex	x	x
SecurityAccess - 27 hex		x
CommunicationControl - 28 hex		x
TesterPresent - 3E hex	x	x
AccessTimingParameter - 83 hex		x
SecuredDataTransmission - 84 hex		x
ControlDTCSetting - 85 hex		x
ResponseOnEvent - 86 hex	x[1)]	x
LinkControl - 87 hex		x
ReadDataByIdentifier - 22 hex	x[2)]	x
ReadMemoryByAddress - 23 hex	x[3)]	x
ReadScalingDataByIdentifier - 24 hex	x[2)]	x
ReadDataByPeriodicIdentifier - 2A hex		x
DynamicallyDefineDataIdentifier - 2C hex	x[4)]	x
WriteDataByIdentifier - 2E hex	x[2)]	x
WriteMemoryByAddress - 3D hex	x[3)]	x
ClearDiagnosticInformation - 14 hex	x	x
ReadDTCInformation - 19 hex	x	x
InputOutputControlByIdentifier - 2F hex	-	x
RoutineControl - 31 hex	x[5)]	x
RequestDownload - 34 hex	-	x
RequestUpload - 35 hex	-	x
TransferData - 36 hex	-	x
RequestTransferExit - 37 hex	-	x

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

8.1.2 Request message definition

Table 23 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**DiagnosticSessionControl Request Service Id**	**M**	**10**	**STDS**
#2	sub-function = [diagnosticSessionType]	M	00-FF	LEV_ DS_

8.1.2.1 Request message sub-function parameter $Level (LEV_) definition

The sub-function parameter diagnosticSessionType is used by the DiagnosticSessionControl service to select the specific behavior of the server. Explanations and usage of the possible diagnostic sessions are detailed below. The following sub-parameter values are specified (responseRequiredIndicationBit (bit 7) not shown):

Table 24 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**reservedByDocument** This value is reserved by this document.	M	RBD
01	**defaultSession** This diagnostic session enables the default diagnostic session in the server(s) and does not support any diagnostic application timeout handling provisions (e.g. no TesterPresent service is necessary to keep the session active). If any other session than the defaultSession has been active in the server and the defaultSession is once again started, then the following implementation rules shall be followed (see also the server diagnostic session state diagram given above): — The server shall stop the current diagnostic session when it has sent the DiagnosticSessionControl positive response message and shall start the newly requested diagnostic session afterwards. — If the server has sent a DiagnosticSessionControl positive response message it shall have re-locked the server if the client unlocked it during the diagnostic session. — If the server sends a negative response message with the DiagnosticSessionControl request service identifier the active session shall be continued. Note, that in case the used data link requires an initialization step then the initialized server(s) shall start the default diagnostic session by default. No DiagnosticSessionControl with diagnosticSession set to defaultSession shall be required after the initialization step.	U	DS

Table 24 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
02	**ProgrammingSession** This diagnosticSession enables all diagnostic services required to support the memory programming of a server. The programmingSession shall only be left via an ECUReset (11 hex) service initiated by the client, a DiagnosticSessionControl (10 hex) service with sessionType equal to defaultSession, or a session layer timeout in the server. In case the server runs in the boot software when it receives the DiagnosticSessionControl (10 hex) service with sessionType equal to defaultSession or a session layer timeout occurs and a valid application software is present for both cases then the server shall restart the application software. This document does not specify the various implementation methods of how to achieve the restart of the valid application software (e.g. a valid application software can be determined directly in the boot software, during the ECU startup phase when performing an ECU reset, etc.).	U	PRGS
03	**extendedDiagnosticSession** This diagnosticSession can e.g. be used to enable all diagnostic services required to support the adjustment of functions like "Idle Speed, CO Value, etc." in the server's memory. It can also be used to enable diagnostic services, which are not specifically tied to the adjustment of functions.	U	EXTDS
04 - 3F	**reservedByDocument** This value is reserved by this document for future definition.	U	RBD
40 - 5F	**vehicleManufacturerSpecific** This range of values is reserved for vehicle manufacturer specific use.	U	VMS
60 - 7E	**systemSupplierSpecific** This range of values is reserved for system supplier specific use.	U	SSS
7F	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

8.1.2.2 Request message data parameter definition

This service does not support data parameters in the request message.

8.1.3 Positive response message definition

Table 25 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**DiagnosticSessionControl Response Service Id**	**S**	**50**	**STDSPR**
#2	diagnosticSessionType	M	00-FF	DS_
#3 : #n	sessionParameterRecord[] #1 = [data#1 : data#m]	 C : C	 00-FF : 00-FF	SPREC_ DATA_1 : DATA_m
C: The presence, structure and content of the sessionParameterRecord is data link layer dependend and therefore defined in the implementation specification(s) of ISO 14229-1				

8.1.3.1 Positive response message data parameter definition

Table 26 — Response message data parameter definition

Definition
diagnosticSessionType This parameter is an echo of the sub-function parameter from the request message.
sessionParameterRecord This parameter record contains session specific parameter values reported by the server. The content and structure of this parameter record is data link layer specific and can be found in the implementation specification(s) of ISO 14229-1.

8.1.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 27 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** Send if the sub-function parameter in the request message is not supported	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This code shall be returned if the criteria for the request DiagnosticSessionControl are not met.	M	CNC

8.1.5 Message flow example(s) DiagnosticSessionControl

8.1.5.1 Example #1 - Start programmingSession

This message flow shows how to enable the diagnostic session "programmingSession" in a server. The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'. For the given example it is assumed that the sessionParameterRecord is supported for the data link layer the service is implemented for.

Table 28 — DiagnosticSessionControl request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DiagnosticSessionControl request SID	10	STDS
#2	diagnosticSessionType = programmingSession, responseRequired=yes	02	DS_ECUPRGS

Table 29 — DiagnosticSessionControl positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DiagnosticSessionControl response SID	50	STDSPR
#2	diagnosticSessionType = programmingSession	02	DS_ECUPRGS

8.2 EcuReset (11 hex) service

The EcuReset service is used by the client to request a server reset.

8.2.1 Service description

This service requests the server to effectively perform a server reset based on the content of the resetType parameter value embedded in the ECUReset request message. The ECUReset positive response message (if required) shall be sent before the reset is executed in the server(s). After a successful server reset the server shall activate the defaultSession.

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

8.2.2 Request message definition

Table 30 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**EcuReset Request Service Id**	**M**	**11**	**ER**
#2	sub-function = [resetType]	M	00-FF	LEV_ RT_

8.2.2.1 Request message sub-function Parameter $Level (LEV_) Definition

The sub-function parameter resetType used by the EcuReset request message to describe how the server has to perform the reset (responseRequiredIndicationBit (bit 7) not shown).

Table 31 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**reservedByDocument** This value is reserved by this document.	M	RBD
01	**hardReset** This value identifies a "hard reset" condition which simulates the power-on / start-up sequence typically performed after a server has been previously disconnected from its power supply (i.e. battery). The performed action is implementation specific and not defined by the standard. It might result in the re-initialization of both volatile memory and non-volatile memory locations to predetermined values.	U	HR
02	**keyOffOnReset** This value identifies a condition similar to the driver turning the ignition key off and back on. This reset condition should simulate a key-off-on sequence (i.e. interrupting the switched power supply). The performed action is implementation specific and not defined by the standard. Typically the values of non-volatile memory locations are preserved; volatile memory will be initialized.	U	KOFFONR
03	**softReset** This value identifies a "soft reset" condition, which causes the server to immediately restart the application program if applicable. The performed action is implementation specific and not defined by the standard. A typical action is to restart the application without reinitializing of previously learned configuration data, adaptive factors and other long-term adjustments.	U	SR

Table 31 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
04	**enableRapidPowerShutDown** This value requests the server to enable and perform a "rapid power shut down" function. The server shall execute the function immediately after "key/ignition" is switched off. While the server executes the power down function, it shall transients either directly or after a defined stand-by-time to sleep mode. If the client requires a response message and the server is already prepared to execute the "rapid power shut down" function, the server shall send the positive response message prior to the start of the "rapid power shut down" function. The next occurrence of a "key on" or "ignition on" signal terminates the "rapid power shut down" function. The client shall not send any request messages other than the ECUReset with the sub-function disableRapidPowerShutDown in order to not disturb the rapid power shut down function. Note: This sub-function is only applicable to a server supporting a stand-by-mode!	U	RPSD
05	**disableRapidPowerShutDown** This value requests the server to disable the previously enabled "rapid power shut down" function.	U	DRPSD
06 - 3F	**reservedByDocument** This range of values is reserved by this document for future definition.	M	RBD
40 - 5F	**vehicleManufacturerSpecific** This range of values is reserved for vehicle manufacturer specific use.	U	VMS
60 - 7E	**systemSupplierSpecific** This range of values is reserved for system supplier specific use.	U	SSS
7F	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

8.2.2.2 Request message data parameter definition

This service does not support data parameters in the request message.

8.2.3 Positive response message definition

Table 32 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**EcuReset Response Service Id**	**S**	**51**	**ERPR**
#2	resetType	M	00-FF	RT_
#3	powerDownTime	C	00-FF	PDT
C: This parameter is present if the sub-function parameter is set to the enableRapidPowerShutDown value (04hex);				

8.2.3.1 Positive response message data parameter definition

Table 33 — Response message data parameter definition

Definition
resetType This parameter is an echo of the sub-function parameter from the request message.
powerDownTime This parameter indicates to the client the minimum time of the stand-by-sequence the server will remain in the power down sequence. The resolution of this parameter is one (1) secound per count. The follwing values are valid: — 00 – FE hex: 0 – 254 seconds powerDownTime, — FF hex: indicates a failure or time not available.

8.2.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 34 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** Send if the sub-function parameter in the request message is not supported	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This code shall be returned if the criteria for the EcuReset request is not met.	M	CNC

8.2.5 Message flow example ECUReset

This section specifies the conditions for the example to be fulfilled to successfully perform an ECUReset service in the server.

Condition of server: ignition = on, system shall not be in an operational mode (e.g. if the system is an engine management, engine shall be off);

The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

NOTE The server shall send an ECUReset positive response message before the server performs the resetType.

Table 35 — EcuReset request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	EcuReset request SID	11	ER
#2	resetType = hardReset, responseRequired=yes	01	RT_HR

Table 36 — EcuReset positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	EcuReset response SID	51	ERPR
#2	resetType = hardReset	01	RT_HR

8.3 SecurityAccess (27 hex) service

The purpose of this service is to provide a means to access data and/or diagnostic services, which have restricted access for security, emissions, or safety reasons. Diagnostic services for downloading/uploading routines or data into a server and reading specific memory locations from a server are situations where security access may be required. Improper routines or data downloaded into a server could potentially damage the electronics or other vehicle components or risk the vehicle's compliance to emission, safety, or security standards.

8.3.1 Service description

The security concept uses a seed and key relationship.

A typical example of the use of this service is as follows:

— client requests the "Seed"

— server sends the "Seed"

— client sends the "Key" (appropriate for the Seed received)

— server responds that the "Key" was valid and that it will unlock itself

A vehicle manufacturer specific time delay (can be zero (0)) might be required before the server can positively respond to a service SecurityAccess 'requestSeed' message from the client after server power up/reset. If a delay timer is supported then this delay shall be activated after a failed SecurityAccess service attempt (see further description below) and when the server is powered up/reset and a previously performed SecurityAccess service has failed. In case the server supports a delay timer then after a successful SecurityAccess service execution the server internal indication information for a delay timer invocation on a power up/reset shall be cleared by the server. In case the server supports a delay timer and cannot determine if the last SecurityAccess service prior to the power up/reset has failed then the delay timer shall always be active after power up/reset. The delay is only required if the server is locked when powered up/reset. The vehicle manufacturer shall select if the delay timer is supported (see annex G.1 for delay timer value examples).

The client shall request the server to "unlock" by sending the service SecurityAccess 'requestSeed' message. The server shall respond by sending a "seed" using the service SecurityAccess 'requestSeed' positive response message. The client shall then respond by returning a "key" number back to the server using the service SecurityAccess 'sendKey' request message. The server shall compare this "key" to one internally stored/calculated. If the two numbers match, then the server shall enable ("unlock") the client's access to specific services/data and indicate that with the service SecurityAccess 'sendKey' positive response message. Vehicle manufacturer may choose to implement a delay after a certain number of failed attempts. An invalid key requires the client to start over from the beginning with a SecurityAccess request message.

If a server supports security, but is already unlocked when a SecurityAccess 'requestSeed' message is received, that server shall respond with a SecurityAccess 'requestSeed' positive response message service with a seed value equal to zero (0). The client shall use this method to determine if a server is locked by checking for a non-zero seed.

Attempts to access security shall not prevent normal vehicle communications or other diagnostic communication.

Servers, which provide security shall support reject messages if a secure service is requested while the server is locked.

Some diagnostic functions/services requested during a specific diagnostic session may require a successful security access sequence. In such case the following sequence of services shall be required:

— DiagnosticSessionControl service

— SecurityAccess service

— Secured diagnostic service

There are different accessModes allowed for an enabled diagnosticSession (session started) in the server.

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

8.3.2 Request message definition

Table 37 — Request message definition - sub-function = requestSeed

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**SecurityAcces Request Service Id**	**M**	**27**	**SA**
#2	sub-function = [securityAccessType = requestSeed]	M	01, 03, 05, 07-7D	LEV_ SAT_RSD
#3 : #n	securityAccessDataRecord[] = [parameter#1 : parameter#m]	C/U : C/U	00-FF : 00-FF	SECACCDR_ PARA1 : PARAm
C: The presence of this parameter depends on the sub-function parameter. It is user optional to be present if the securityAccessType parameter indicates that the client requests a seed from the server (requestSeed).				

Table 38 — Request message definition - sub-function = sendKey

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**SecurityAcces Request Service Id**	**M**	**27**	**SA**
#2	sub-function = [securityAccessType = sendKey]	M	02, 04, 06, 08-7E	LEV_ SAT_SK
#3 : #n	securityKey[] = [key#1 (high byte) : key#m (low byte)]	C : C	00-FF : 00-FF	SECKEY_ KEY1HB : KEYmLB
C: The presence of this parameter depends on the sub-function parameter. It is mandatory to be present if the securityAccessType parameter indicates that the client transmits a key to the server (sendKey).				

8.3.2.1 Request message sub-function parameter $Level (LEV_) definition

The sub-function parameter securityAccessType indicates to the server the step in progress for this service, the level of security the client wants to access and the format of seed and key. If a server supports different levels of security each level shall be identified by the requestSeed value, which has a fixed relationship to the sendKey value.

Examples:

— “requestSeed=01 hex” identifies a fixed relationship between “requestSeed=01 hex” and “sendKey=02 hex”

— “requestSeed=03 hex” identifies a fixed relationship between “requestSeed=03 hex” and “sendKey=04 hex”

Values are defined in the table below for requestSeed and sendKey (responseRequiredIndicationBit (bit 7) not shown).

Table 39 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**reservedByDocument** This value is reserved by this document.	M	RBD
01	**requestSeed** RequestSeed with the level of security defined by the vehicle manufacturer.	U	RSD
02	**sendKey** SendKey with the level of security defined by the vehicle manufacturer.	U	SK
03, 05, 07-3F	**requestSeed** RequestSeed with different levels of security defined by the vehicle manufacturer.	U	RSD
04, 06, 08-40	**sendKey** SendKey with different levels of security defined by the vehicle manufacturer.	U	SK
41 - 5F	**vehicleManufacturerSpecific** This range of values is reserved for vehicle manufacturer specific use.	U	VMS
60 - 7E	**systemSupplierSpecific** This range of values is reserved for system supplier specific use.	U	SSS
7F	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

8.3.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 40 — Request message data parameter definition

Definition
securityKey (high and low bytes) The “Key” parameter in the request message is the value generated by the security algorithm corresponding to a specific “Seed” value.
securityAccessDataRecord This parameter record is user optionally to transmit data to a server when requesting the seed information. It can e.g. contain an identification of the client that is verified in the server.

8.3.3 Positive response message definition

Table 41 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**SecurityAccess Response Service Id**	**S**	**67**	**SAPR**
#2	securityAccessType	M	00-FF	SAT_
#3 : #n	securitySeed[] = [seed#1 (high byte) : seed#m (low byte)]	C : C	00-FF : 00-FF	SECSEED_ SEED1HB : SEEDmLB
C: The presence of this parameter depends on the securityAccessType parameter. It is mandatory to be present if the securityAccessType parameter indicates that the client wants to retrieve the seed from the server.				

8.3.3.1 Positive response message data parameter definition

Table 42 — Response message data parameter definition

Definition
securityAccessType This parameter is an echo of the sub-function parameter from the request message.
securitySeed (high and low bytes) The seed parameter is a data value sent by the server and is used by the client when calculating the key needed to access security. The securitySeed data bytes are only present in the response message if the request message was sent with the sub-parameter set to a value which requests the seed of the server.

8.3.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 43 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** Send if the sub-function parameter in the request message is not supported	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This code shall be returned if the criteria for the request SecurityAccess are not met.	M	CNC
24	**requestSequenceError** Send if the 'sendKey' sub-parameter is received without first receiving a 'requestSeed' request message.	M	RSE
31	**requestOutOfRange** This code shall be sent if the user optional securityAccessDataRecord contains invalid data.	M	ROOR
35	**invalidKey** Send if the value of the key is not valid for the server.	M	IK

Table 43 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
36	**ExceededNumberOfAttempts** Send if too many attempts with invalid values are requested.	M	ENOA
37	**RequiredTimeDelayNotExpired** Send if the delay timer is active and a request is transmitted.	M	RTDNE

8.3.5 Message flow example(s) SecurityAccess

For the below given message flow examples the following conditions have to be fulfilled to successfully unlock the server if it is in a “locked” state:

— sub-parameter to request the seed: 01 hex (requestSeed)

— sub-parameter to send the key: 02 hex (sendKey)

— seed of the server (2 bytes): 3657 hex

— key of the server (2 bytes): C9A9 hex (e.g. 2’s complement of the seed value)

The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

8.3.5.1 Example #1 - server is in a “locked” state

8.3.5.1.1 Step #1: Request the Seed

Table 44 — SecurityAccess request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	SecurityAccess request SID	27	SA
#2	securityAccessType = requestSeed, responseRequired = yes	01	SAT_RSD

Table 45 — SecurityAccess positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	SecurityAccess response SID	67	SAPR
#2	securityAccessType = requestSeed	01	SAT_RSD
#3	securitySeed [byte 1] = seed #1 (high byte)	36	SECHB
#4	securitySeed [byte 2] = seed #2 (low byte)	57	SECLB

8.3.5.1.2 Step #2: Send the Key

Table 46 — SecurityAccess request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	SecurityAccess request SID	27	SA
#2	securityAccessType = sendKey, responseRequired = yes	02	SAT_SK
#3	securityKey [byte 1] = key #1 (high byte)	C9	SECKEY_HB
#4	securityKey [byte 2] = key #2 (low byte)	A9	SECKEY_LB

Table 47 — SecurityAccess positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	SecurityAccess response SID	67	SAPR
#2	securityAccessType = sendKey	02	SAT_SK

8.3.5.2 Example #2 - server is in an “unlocked” state

8.3.5.2.1 Step #1: Request the Seed

Table 48 — SecurityAccess request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	SecurityAccess request SID	27	SA
#2	securityAccessType = requestSeed, responseRequired = yes	01	SAT_RSD

Table 49 — SecurityAccess positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	SecurityAccess response SID	67	SAPR
#2	securityAccessType = requestSeed	01	SAT_RSD
#3	securitySeed [byte 1] = seed #1 (high byte)	00	SECHB
#4	securitySeed [byte 2] = seed #2 (low byte)	00	SECLB

8.4 CommunicationControl (28 hex) service

The purpose of this service is to switch on/off the transmission and/or the reception of certain messages of (a) server(s) (e.g. application communication messages).

8.4.1 Service description

The server shall perform the requested communication type control after sending the CommunicationControl positive response message to the client (responseRequired = yes). In case no response is requested from the client then the server shall perform the requested communication type control immediately after the successful evaluation of the request message.

The control of the message transmission via this service overrides the responseRequiredIndicationBit embedded in the sub-function parameter of all consecutive ISO 14229-1 diagnostic services utilizing a sub-function parameter.

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

IMPORTANT — If the sub-function parameter includes the 'enableRxAndTx' data value and 'communicationType = diagnosticCommunicationMessages' and the 'responseRequiredIndicationBit includes responseRequired = NO' then no response message shall be sent by the server. If the sub-function parameter includes the 'enableRxAndDisableTx' data value and 'communicationType = diagnosticCommunicationMessages' and the responseRequiredIndicationBit includes 'responseRequired = YES' then no response message shall be sent by the server.

8.4.2 Request message definition

Table 50 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**CommunicationControl Request Service Id**	**M**	**28**	**CC**
#2	sub-function = [controlType]	M	00-FF	LEV_ CTRLTP
#3	communicationType	M	00-FF	CTP

8.4.2.1 Request message sub-function parameter $Level (LEV_) definition

The sub-function parameter controlType contains information on how the server shall modify the communication type referenced in the communicationType parameter (responseRequiredIndicationBit (bit 7) not shown in the table below).

Table 51 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**enableRxAndTx** This value indicates that the reception and transmission of messages shall be enabled for the specified communicationType.	U	ERXTX
01	**enableRxAndDisableTx** This value indicates that the reception of messages shall be enabled and the transmission shall be disabled for the specified communicationType.	U	ERXDTX
02	**disableRxAndEnableTx** This value indicates that the reception of messages shall be disabled and the transmission shall be enabled for the specified communicationType.	U	DRXETX
03	**disableRxAndTx** This value indicates that the reception and transmission of messages shall be disabled for the specified communicationType.	U	DRXTX
04 - 3F	**reservedByDocument** This range of values is reserved by this document for future definition.	U	RBD
40 - 5F	**vehicleManufacturerSpecific** This range of values is reserved for vehicle manufacturer specific use.	U	VMS
60 - 7E	**systemSupplierSpecific** This range of values is reserved for system supplier specific use.	U	SSS
7F	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

8.4.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 52 — Request message data parameter definition

communicationType This parameter is used to reference the kind of communication to be controlled. The communicationType parameter is a bit-code value, which allows controlling multiple communication types at the same time. (see annex B.1 for the coding of the communicationType data parameter) Note, that disabling the reception of diagnostic messages is not permitted and shall therefore be rejected by the server (see table below with negative response codes for this service).

8.4.3 Positive response message definition

Table 53 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**CommunicationControl Response Service Id**	**S**	**68**	**CCPR**
#2	controlType	M	00-FF	CTRLTP

8.4.3.1 Positive response message data parameter definition

Table 54 — Response message data parameter definition

Definition
controlType
This parameter is an echo of the sub-function parameter from the request message.

8.4.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 55 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** Send if the sub-function parameter in the request message is not supported	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** Used when the server is in a critical normal mode activity and therefore cannot disable/enable the requested communication type.	M	CNC
31	**requestOutOfRange** 1. The server shall use this response code, if it detects an error in the communicationType parameter. 2. The server shall use this response code if it detects that the client wants to disable the reception of diagnostic messages.	M	ROOR

8.4.5 Message flow example CommunicationControl (disable transmission of network management messages)

The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

Table 56 — CommunicationControl request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	CommunicationControl request SID	28	CC
#2	controlType = enableRxAndDisableTx, responseRequired = yes	01	ERXTX
#3	communicationType = network management	02	NWMCP

Table 57 — CommunicationControl positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	CommunicationControl response SID	68	CCPR
#2	controlType	01	CTRLTP

8.5 TesterPresent (3E hex) service

This service is used to indicate to a server (or servers) that a client is still connected to the vehicle and that certain diagnostic services and/or communication that have been previously activated are to remain active.

8.5.1 Service description

This service is used to keep one or multiple servers in a diagnostic session other than the defaultSession. This can either be done by transmitting the TesterPresent request message periodically or in case of the absence of other diagnostic services to prevent the server(s) from automatically returning to the defaultSession. The detailed session requirements that apply to the use of this service when keeping a single server or multiple servers in a diagnostic session other than the defaultSession can be found in the implementation specifications of ISO 14229-1.

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

8.5.2 Request message definition

Table 58 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**TesterPresent Request Service Id**	**M**	**3E**	**TP**
#2	sub-function = [zeroSubFunction]	M	00/80	LEV_ ZSUBF

8.5.2.1 Request message sub-function parameter $Level (LEV_) definition

The table below specifies the sub-function parameter values defined for this service (responseRequiredIndicationBit (bit 7) not shown).

Table 59 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**zeroSubFunction** This parameter value is used to indicate that no sub-function value beside the responseRequiredIndicationBit is supported by this service.	M	ZSUBF
01 - 7F	**reservedByDocument** This range of values is reserved by this document.	M	RBD

8.5.2.2 Request message data parameter definition

This service does not support data parameters in the request message.

8.5.3 Positive response message definition

Table 60 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**TesterPresent Response Service Id**	**S**	**7E**	**TPPR**
#2	zeroSubFunction	M	00	ZSUBF

8.5.3.1 Positive response message data parameter definition

Table 61 — Response message data parameter definition

Definition
zeroSubFunction This parameter is an echo of the sub-function parameter from the request message.

8.5.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 62 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** Send if the sub-function parameter in the request message is not supported	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF

8.5.5 Message flow example(s) TesterPresent

8.5.5.1 Example #1 - TesterPresent (responseRequired = yes)

Table 63 — TesterPresent request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TesterPresent request SID	3E	TP
#2	zeroSubFunction, responseRequired = yes	00	ZSUBF

Table 64 — TesterPresent positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TesterPresent response SID	7E	TPPR
#2	zeroSubFunction	00	ZSUBF

8.5.5.2 Example #2 - TesterPresent (responseRequired = no)

Table 65 — TesterPresent request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TesterPresent request SID	3E	TP
#2	zeroSubFunction, responseRequired = no	80	ZSUBF

There is no response sent by the server(s).

8.6 AccessTimingParameter (83 hex) service

The AccessTimingParameter service is used to read and change the default timing parameters of a communication link for the duration this communication link is active.

8.6.1 Service description

The use of this service is complex and depends on the server's capability and the data link topology. Only one extended timing parameter set will be supported per diagnostic session. It is recommended to use this service only with physical addressing, because of the different sets of extended timing parameters supported by the servers.

It is recommended to use the following sequence of services

— DiagnosticSessionControl (diagnosticSessionTyp) service

— AccessTimingParameter (readExtendedTimingParameterSet) service

— AccessTimingParameter (setTimingParametersToGivenValues) service

For the case a response is required to be sent by the server the client and server shall activate the new timing parameter settings after the server has sent the AccessTimingParameter positive response message. In case no response message is allowed the client and the server shall activate the new timing parmeter after the transmission/reception of the request message.

The server and the client shall reset their timing parameters to the default values after a successful switching to another or the same diagnostic session (e.g. via DiagnosticSessionControl, ECUReset service, or a session timing timeout).

The AccessTimingParameter service provides four (4) different modes for the access to the server timing parameters:

— readExtendedTimingParameterSet

— setTimingParametersToDefaultValues

— readCurrentlyActiveTimingParameters

— setTimingParametersToGivenValues

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

8.6.2 Request message definition

Table 66 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**AccessTimingParameter Request Service Id**	**M**	**83**	**ATP**
#2	sub-function = [timingParameterAccessType]	M	00-FF	LEV_ TPAT_
#3 : #n	TimingParameterRequestRecord [byte #1 : byte #m]	C : C	00-FF : 00-FF	TPREQR_ B1 : Bm

Table 66 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
C: The TimingParameterRequestRecord is only present if timingParameterAccessType = setTimingParametersToGivenValues. The structure and content of the TimingParameterRequestRecord is data link layer dependend and therefore defined in the implementation specification(s) of ISO 14229-1.				

8.6.2.1 Request message sub-function parameter $Level (LEV_) definition

The sub-function parameter timingParameterAccessType is used by the AccessTimingParameter service to select the specific behavior of the server. Explanations and usage of the possible timingParameterIdentifiers are detailed below. The following sub-parameter values are specified (responseRequiredIndicationBit (bit 7) not shown):

Table 67 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**reservedByDocument** This value is reserved by this document.	M	RBD
01	**readExtendedTimingParameterSet** Upon receiving an AccessTimingParameter indication primitive with timingParameterAccessType = readExtendedTimingParameterSet, the server shall read the extended timing parameter set, i.e. the values that the server is capable of supporting. If the read access to the timing parameter set is successful, the server shall send an AccessTimingParameter response primitive with the positive response parameters. If the read access to the timing parameters set is not successful, the server shall send a negative response message with the appropriate negative response code. This subfunction is used to provide an extra set of timing parameters for the currently active diagnostic session. With the timingParameterAccessType = setTimingParametersToGivenValues only this set (read by timingParameterAccessType = readExtendedTimingParameterSet) of timing parameters can be set.	U	RETPS
02	**setTimingParametersToDefaultValues** Upon receiving an AccessTimingParameter indication primitive with timingParameterAccessType = setTimingParametersToDefaultValues, the server shall change all timing parameters to the default values and send an AccessTimingParameter response primitive with the positive response parameters before the default timing parameters become active (if responseRequired is set to 'yes', otherwise the timing parameters shall become active after the successful evaluation of the request message). If the timing parameters cannot be changed to default values for any reason, the server shall maintain the currently active timing parameters and send a negative response message with the appropriate negative response code. The definition of the default timing values depends on the used data link and is specified in the implementation specification(s) of ISO 14229-1.	U	STPTDV
03	**readCurrentlyActiveTimingParameters** Upon receiving an AccessTimingParameter indication primitive with timingParameterAccessType = readCurrentlyActiveTimingParameters, the server shall read the currently used timing parameters. If the read access to the timing parameters is successful, the server shall send an AccessTimingParameter response primitive with the positive response parameters. If the read access to the currently used timing parameters is impossible for any reason, the server shall send a negative response message with the appropriate negative response code.	U	RCATP

Table 67 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
04	**setTimingParametersToGivenValues** Upon receiving an AccessTimingParameter indication primitive with timingParameterAccessType = setTimingParametersToGivenValues, the server shall check if the timing parameters can be changed under the present conditions. If the conditions are valid, the server shall perform all actions necessary to change the timing parameters and send an AccessTimingParameter response primitive with the positive response parameters before the new timing parameter values become active (if responseRequired is set to 'yes', otherwise the timing parameters shall become active after the successful evaluation of the request message). If the timing parameters cannot be changed by any reason, the server shall maintain the currently active timing parameters and send a negative response message with the appropriate negative response code. It is not possible to set the timing parameters of the server to any set of values between the minimum and maximum values read via timingParameterAccessType = readExtendedTimingParameterSet. The timing parameters of the server can only be set to exactly the timing parameters read via timingParameterAccessType = readExtendedTimingParameterSet. A request to do so shall be rejected by the server.	U	STPTGV
05-FF	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

8.6.2.2 Request message data parameter definition

The following data-parameters are defined for the request message:

Table 68 —Request message data parameter definition

Definition
TimingParameterRequestRecord This parameter record contains the timing parameter values to be set in the server via timingParameterAccessType = setTimingParametersToGivenValues. The content and structure of this parameter record is data link layer specific and can be found in the implementation specification(s) of ISO 14229-1.

8.6.3 Positive response message definition

Table 69 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**AccessTimingParameter Response Service Id**	**S**	**C3**	**ATPPR**
#2	timingParameterAccessType	M	00-FF	TPAT_
#3 : #n	TimingParameterResponseRecord [byte #1 : byte #m]	C : C	00-FF : 00-FF	TPRSPR_ B1 : Bm
C: The TimingParameterResponseRecord is only present if timingParameterAccessType = readExtendedTimingParameterSet or readCurrentlyActiveTimingParameters. The structure and content of the TimingParameterResponseRecord is data link layer dependend and therefore defined in the implementation specification(s) of ISO 14229-1.				

8.6.3.1 Positive response message data parameter definition

Table 70 — Response message data parameter definition

Definition
timingParameterAccessType This parameter is an echo of the sub-function parameter from the request message.
TimingParameterResponseRecord This parameter record contains the timing parameter values read from the server via timingParameterAccessType = readExtendedTimingParameterSet or readCurrentlyActiveTimingParameters. The content and structure of this parameter record is data link layer specific and can be found in the implementation specification(s) of ISO 14229-1.

8.6.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 71 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** Send if selected Timing Parameter Access Type is not supported	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message or the format is wrong.	M	IMLOIF
22	**conditionsNotCorrect** This code shall be returned if the criteria for the request AccessTimingParameter are not met.	M	CNC
31	**requestOutOfRange** This code shall be sent if the TimingParameterRequestRecord contains invalid timing parameter values.	M	ROOR

8.6.5 Message flow example(s) AccessTimingParameter

8.6.5.1 Example #1 – set timing parameters to default values

This message flow shows how to set the default timing parameters in a server. The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

Table 72 — AccessTimingParameter request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	AccessTimingParameter request SID	83	ATP
#2	timingParameterAccessType = setTimingParametersToDefaultValues responseRequired=yes	02	TPAT_STPTDV

Table 73 — AccessTimingParameter positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	AccessTimingParameter response SID	C3	ATPPR
#2	timingParameterAccessType = setTimingParametersToDefaultValues	02	TPAT_STPTDV

Further examples for the usage of this service can be found in the implementation specifications of ISO 14229-1.

8.7 SecuredDataTransmission (84 hex) service

The purpose of this service is to transmit data that is protected against attacks from third parties - which could endanger data security - according to ISO 15764.

8.7.1 Service description

The SecuredDataTransmission service is applicable if a client intends to use diagnostic services defined in this document in a secured mode. It may also be used to transmit external data, which conform to some other application protocol, in a secured mode between a client and a server. A secured mode in this context means that the data transmitted is protected by cryptographic methods.

8.7.1.1 Security sub-layer

This section briefly describes the security sub-layer as defined in ISO 15764 (see ISO 15764 for details).

The following figure illustrates the security sub-Layer as defined in ISO 15764. The security sub-layer has to be added in the server and client application for the purpose of performing diagnostic services in a secured mode.

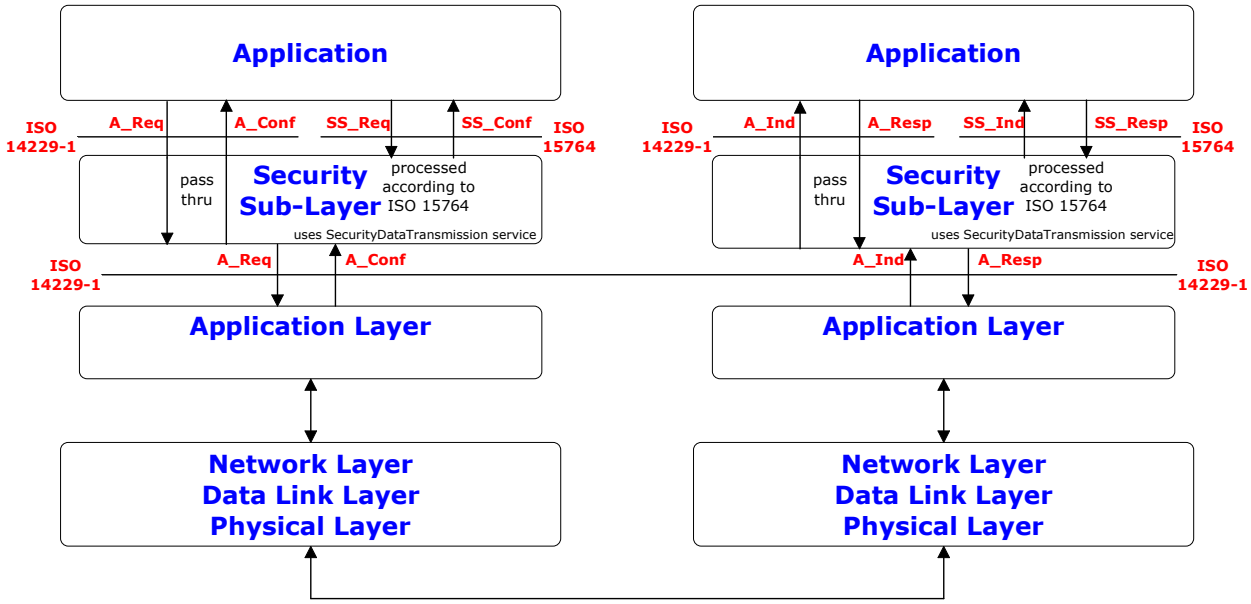


Figure 9 — Security Sub-Layer implementation

There are two (2) methods to perform diagnostic service data transfer between the client and server(s):

— Unsecured data transmission mode

The application uses the diagnostic Services and Application Layer Service Primitives defined in this document to exchange data between a client and a server. The Security Sub-Layer performs a "Pass-Thru" of data between "Application" and "Application Layer" in the client and the server.

— Secured data transmission mode

The application uses the diagnostic Services or external services and the Security Sub-Layer Service Primitives defined in ISO 15764 to exchange data between a client and a server. The security sub-layer uses the SecuredDataTransmission service for the transmission/reception of the secured data. Secured links must be point-to-point communication. Therefore only physical addressing is allowed, which means that only one server is involved.

The interface of the security sub-layer to the application is according to the ISO/OSI model conventions and therefore provides the following four (4) security sub-layer (SS_) service primitives

— SS_SecuredMode.req: Security Sub-Layer Request

— SS_SecuredMode.ind: Security Sub-Layer Indication

— SS_SecuredMode.resp: Security Sub-Layer Response

— SS_SecuredMode.conf: Security Sub-Layer Confirmation

ISO 14229 defines both, confirmed and unconfirmed services. In a secured mode only confirmed services are allowed (responseRequired = yes). Based on this requirement the following services are not allowed to be executed in a secured mode:

— ResponseOnEvent (86 hex)

— ReadDataByPeriodicIdentifier (2A hex)

— TesterPresent (3E hex)

The confirmed services (responseRequired = yes) use the four (4) application layer service primitives request, indication, response and confirmation. Those are mapped onto the four (4) security sub-layer service primitives and vice versa when executing a confirmed diagnostic service in a secured mode.

The task of the Security Sub-Layer when performing a diagnostic service in a secured mode is to encrypt data provided by the "Application", to decrypt data provided by the "Application Layer" and to add, check, and remove security specific data elements. The Security Sub-Layer uses the SecuredDataTransmission (84 hex) service of the application layer to transmit and receive the entire diagnostic message or message according to an external protocol (request and response) which shall be exchanged in a secured mode.

The security sub-layer provides the service "SecuredServiceExecution" to the application for the purpose of a secured execution of diagnostic services.

The security sub-layer request and indication primitive of the "SecuredServiceExecution" service are specified in ISO 15764 according to the following general format:

SS_SecuredMode.request (
SA,
TA,
TA_type,
[RA,]
[,parameter 1, ...]
)

SS_SecuredMode.indication (
SA,
TA,
TA_type,
[RA,]
[,parameter 1, ...]
)

The security sub-layer layer response and confirm primitive of the SecuredServiceExecution service are specified in ISO 15764 according to the following general format:

SS_SecuredMode.response (
SA,
TA,

	TA_type, **RA (optional)** **Result,** **[parameter 1, ...]** **)**
SS_SecuredMode.confirm	**(** **SA,** **TA,** **TA_type,** **RA (optional)** **Result,** **[parameter 1, ...]** **)**

Detailed information about

— the security sub-layer service primitives (Service Data Units (SDU), [parameter 1, ...])

— the security sub-layer protocol data units (PDU)

— the tasks to be performed by the security sub-layer for a secured data transmission

can be found in ISO 15764.

The addressing information shown in the security sub-layer service primitves is mapped directly onto the addressing information of the application layer and vice versa.

8.7.1.2 Security sub-layer access

The concept of accessing the security sub-layer for a secured service execution is similar to the application layer interface as described in this document. The security sub-layer makes use of the application layer service primitives.

The following describes the execution of confirmed diagnostic service in a secured mode:

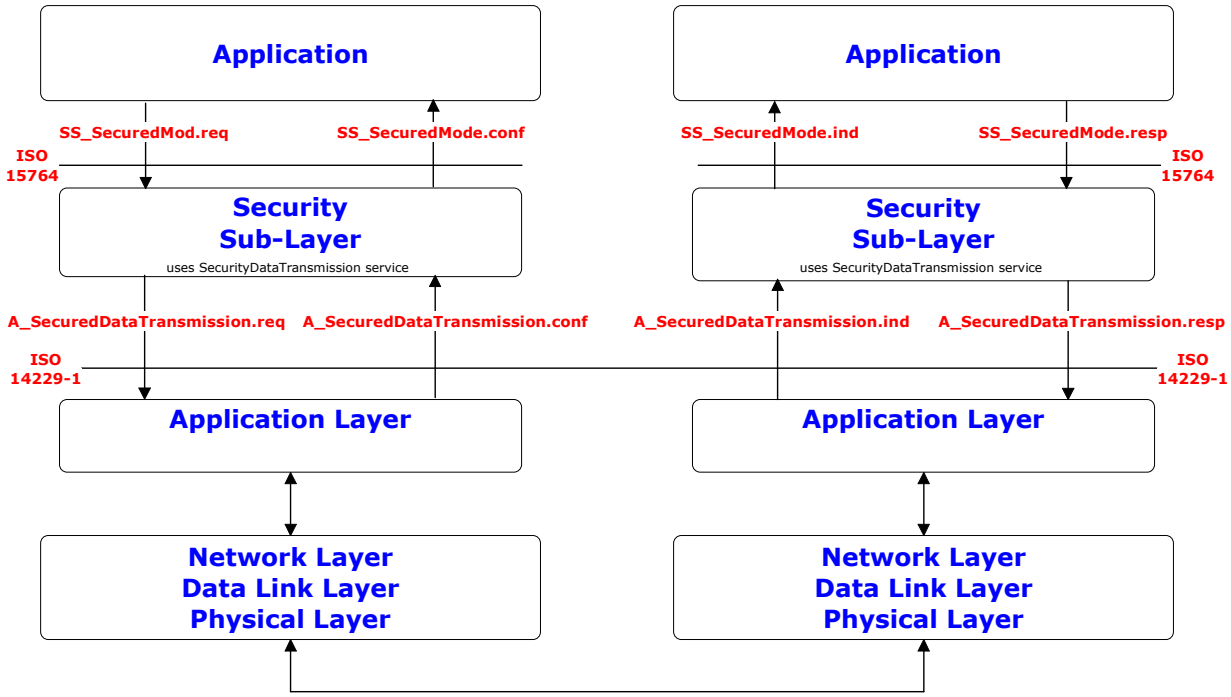
— The client application uses the security sub-layer SecuredServiceExecution service request to perform a diagnostic service in a secured mode. The security sub-layer performs the required action to establish a link with the server(s), adds the specific security related parameters, encrypts the service data of the diagnostic service to be executed in a secured mode if needed and uses the application layer SecuredDataTransmission service request to transmit the secured data to the server.

— The server receives an application layer SecuredDataTransmission service indication, which is handled by the security sub-layer of the server. The security sub-layer of the server checks the security specific parameters, decrypts encrypted data and presents the data of the service to be executed in a secured mode to the application via the security sub-layer SecuredServiceExecution service indiciation. The application executes the service and uses the security sub-layer SecuredServiceExecution service response to respond to the service in a secured mode. The security sub-layer of the server adds the specific security related parameters, encrypts the response message data if needed and uses the application layer SecuredDataTransmission service response to transmit the response data to the client.

— The client receives an application layer SecuredDataTransmission service confirmation primitive, which is handled by the security sub-layer of the client. The security sub-layer of the client checks the security specific parameters, decrypts encrypted response data and presents the data via the security sub-layer SecuredServiceExecution confirmation to the application.

The following figure graphically shows the interaction of the security sub-layer, the application layer, and the application when executing a confirmed diagnostic service in a secured mode.

Figure 10 — Security sub-layer, application layer, and application interaction



8.7.2 Request message definition

The security sub-layer generates the application layer SecuredDataTransmission request message parameters according to the rules defined in ISO 15764.

Table 74 —Request message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**SecuredDataTransmission Request Service Id**	**M**	**84**	**SDT**
#2 : #n	securityDataRequestRecord[ ] = [securityDataParameter#1 : securityDataParameter#m]	M : M	00-FF : 00-FF	SECDRQR_ SDP_ : SDP_

8.7.2.1 Request message sub-Function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

8.7.2.2 Request message data parameter definition

The following data-parameters are defined for the request message:

Table 75 —Request message data parameter definition

Definition
securityDataRequestRecord This parameter contains the data as processed by the Security Sub-Layer and is defined in ISO 15764.

8.7.3 Positive response message definition

Table 76 —Positive response message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
1	**SecuredDataTransmission Response Service Id**	**M**	**C4**	**SDTPR**
2 : n	securityDataResponseRecord[ ] = [securityDataParameter#1 : securityDataParameter#m]	M : M	00-FF : 00-FF	SECDRQR_ SDP_ : SDP_

8.7.3.1 Positive response message data parameter definition

The following data-parameters are defined for the positive response message:

Table 77 — Response message data parameter definition

Definition
securityDataResponseRecord
This parameter contains the data as processed by the Security Sub-Layer and is defined in ISO 15764.

8.7.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below. The response codes are always sent without encryption, even if according to the configurationProfile in the request A_PDU the response A_PDU has to be encrypted.

Table 78 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** The server shall use this response code, if the length of the request A_PDU is not correct.	M	IMLOIF
38 - 4F	**reservedByExtendedDataLinkSecurityDocument** This range of values is reserved by ISO 15764 Extended data link security. Applicable negative response codes are defined in ISO 15764.	M	RBEDLSD

NOTE The response codes listed above apply to the SecuredDataTransmission (84 hex) service. In case the diagnostic service performed in a secured mode requires a negative response then this negative response is send to the client in a secured mode via a SecuredDataTransmission positive response message.

8.8 ControlDTCSetting (85 hex) service

The ControlDTCSetting service shall be used by a client to stop or resume the setting of diagnostic trouble codes (DTCs) in the server(s).

8.8.1 Service description

The ControlDTCSetting request message can be used to stop the setting of diagnostic trouble codes in an individual server or a group of servers. If the server being addressed is not able to stop the setting of diagnostic trouble codes , it shall respond with a ControlDTCSetting negative response message indicating the reason for the reject.

When the setting of DTCs is disabled then the DTC status bits shall reflect the state prior to disabling the DTC setting for the duration the DTCs are disabled. The update of the DTC status bit information shall continue once a ControlDTCSetting request is performed with sub-function set to "on" or a session layer timeout occurs (server transitions to defaultSession).

In case a successful ECUReset is performed then this re-enables the setting of DTCs.

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

8.8.2 Request message definition

Table 79 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ControlDTCSetting Request Service Id**	**M**	**85**	**CDTCS**
#2	sub-function = [DTCSettingType]	M	 00-FF	LEV_ DTCSTP_
 #3 : #n	DTCSettingControlOptionRecord [] = [parameter#1 : parameter#m	 U : U	 00-FF : 00-FF	DTCSCOR_ PARA1 : PARAm

8.8.2.1 Request message sub-function parameter $Level (LEV_) definition

The sub-function parameter DTCSettingType is used by the ControlDTCSetting request message to indicate to the server(s) whether diagnostic trouble code setting shall stop or start again (responseRequiredIndicationBit (bit 7) not shown in the table below).

Table 80 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**reservedByDocument** This value is reserved by this document.	M	RBD
01	**on** The server(s) shall resume the setting of diagnostic trouble codes according to normal operating conditions	M	ON
02	**off** The server(s) shall stop the setting of diagnostic trouble codes.	M	OFF

Table 80 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
03 - 3F	**reservedByDocument** This range of values is reserved by this document for future definition.	M	RBD
40 - 5F	**vehicleManufacturerSpecific** This range of values is reserved for vehicle manufacturer specific use.	U	VMS
60 - 7E	**systemSupplierSpecific** This range of values is reserved for system supplier specific use.	U	SSS
7F	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

8.8.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 81 — Request message data parameter definition

Definition
DTCSettingControlOptionRecord This parameter record is user optionally to transmit data to a server when controlling the DTC setting. It can e.g. contain a list of DTCs to be turned on or off.

8.8.3 Positive response message definition

Table 82 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ControlDTCSetting Response Service Id**	**S**	**C5**	**CDTCSPR**
#2	DTCSettingType	M	00-FF	DTCSTP

8.8.3.1 Positive response message data parameter definition

Table 83 — Response message data Parameter definition

Definition
DTCSettingType This parameter is an echo of the sub-function parameter from the request message.

8.8.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 84 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** Send if the sub-function parameter in the request message is not supported	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** Used when the server is in a critical normal mode activity and therefore cannot perform the requested DTC control functionality.	U	CNC
31	**requestOutOfRange** The server shall use this response code, if it detects an error in the DTCSettingControlOptionRecord.	M	ROOR

8.8.5 Message flow example(s) ControlDTCSetting

8.8.5.1 Example #1 - ControlDTCSetting (DTCSettingType = off)

Note that this example does not use the capability of the service to transfer additional data to the server. The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

Table 85 — ControlDTCSetting request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ControlDTCSetting request SID	85	RDTCS
#2	DTCSettingType = off, responseRequired = yes	02	DTCSTP_OFF

Table 86 — ControlDTCSetting positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ControlDTCSetting response SID	C5	RDTCSPR
#2	DTCSettingType = off	02	DTCSTP_OFF

8.8.5.2 Example #2 - ControlDTCSetting(responseRequired = on)

Note that this example does not use the capability of the service to transfer additional data to the server. The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

Table 87 — ControlDTCSetting request message flow example #2

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ControlDTCSetting request SID	85	ENC
#2	DTCSettingType = on, responseRequired = yes	01	DTCSTP_ON

Table 88 — ControlDTCSetting positive response message flow example #2

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ControlDTCSetting response SID	C5	RDTCSPR
#2	DTCSettingType = on	01	DTCSTP_ON

8.9 ResponseOnEvent (86 hex) service

The ResponseOnEvent service requests a server to start or stop transmission of responses on a specified event.

8.9.1 Service description

This service provides the possibility to automatically execute a diagnostic service in case a specified event occurs in the server. The client specifies the event (including optional event parameters) and the service (including service parameters) to be executed in case the event occurs. See the figure below for a brief overview about the client and server behaviour.

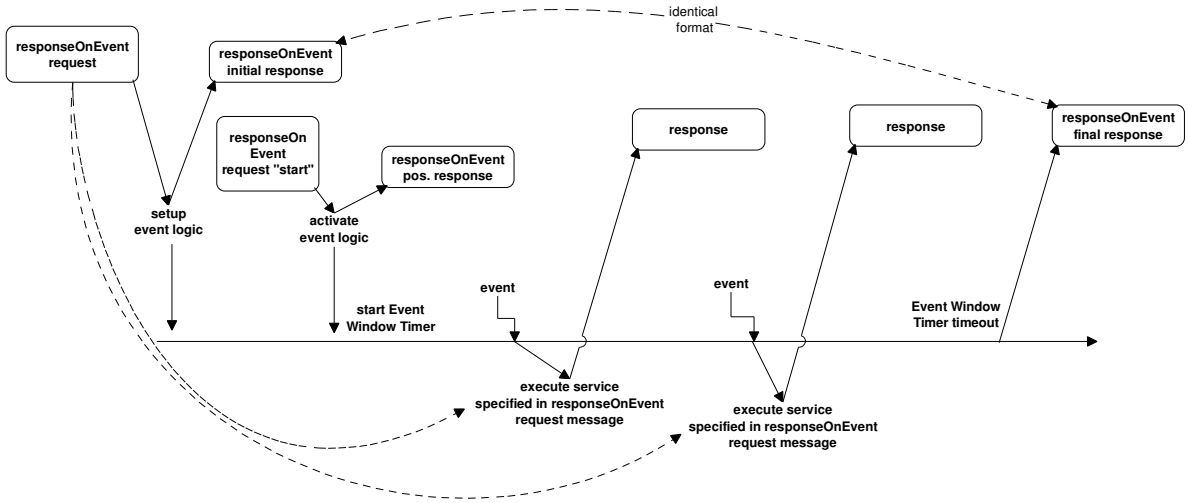


Figure 11 — ResponseOnEvent service - client and server behaviour

NOTE The figure above assumes, that the event window timer is configured to timeout prior to the power down of the server, therefore the final ResponseOnEvent positive response message is shown at the end of the event timing window.

The server shall evaluate the sub-function and data content of the ResponseOnEvent request message at the time of the reception. This includes the following sub-function and parameters:

— eventType,

— eventWindowTime,

— eventTypeRecord (eventTypeParameter #1-#m).

In case of invalid data in the ResponseOnEvent request message a negative response with the negative response code 31 hex shall be sent. The serviceToRespondToRecord is not part of this evaluation. The serviceToRespondToRecord parameter will be evaluated when the specified event occurs, which triggers the execution of the service contained in the serviceToRespondToRecord. At the time the event occurs the serviceToRespondToRecord (diagnostic service request message) shall be executed. In case conditions are not correct a negative response message with the appropriate negative response code shall be sent. Multiple events shall be signalled in the order of their occurrence.

The following implementation rules shall apply:

a) The ResponseOnEvent service can be set up and activated in any session, including the defaultSession. TesterPresent service is not necessarily required to keep the ResponseOnEvent service active.

b) If the specified event occurs when a diagnostic service is in progress, which means that either a request message is in progress to be received, or a request is executed, or a response message is in progress (this includes the negative response message handling with response code 78 hex) to be transmitted (if responseRequired = yes) then the execution of the request message contained in the serviceToRespondToRecord shall be postponed until the completion of the diagnostic service in progress.

c) If the specified event is accepted by the server the client shall not request the following diagnostic services:

   — CommunicationControl,

   — DynamicallyDefineDataIdentifier,

   — RequestDownload,

   — RequestUpload,

   — TransferData,

   — TransferDataExit,

   — RoutineControl.

d) The server is not executing any diagnostic service at the point in time the specified event occurs.

   — server executes the service contained in the serviceToRespondToRecord.

e) Once the ResponseOnEvent service is initiated the server shall support the data link where this service has been submitted while the ResponseOnEvent service is active.

f) A DiagnosticSessionControl service shall stop the ResponseOnEvent service regardless whether a different session than the current session or the same session is activated

It is recommended to use only the services listed in the table below for the service to be performed in case the specified event occurs. (serviceToRespondTo request service Identifier).

Table 89 — Recommended services to be used with the ResponseOnEvent service

Recommended services (ServiceToRespondTo)	RequestService Identifier (SId)	ResponseService Identifier (SId)
readDataByIdentifier	22	62
readDTCInformation	19	59
RoutineControl	31	71
inputOutputControlByIdentifier	2F	6F

It is allowed to run different multiple ResponseOnEvent services at a time and to stop individual serviceToRespondTo services. While no serviceToRespondTo is currently in progress running the server has to handle any additional diagnostic service request.

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

8.9.2 Request message definition

Table 90 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ResponseOnEvent Request Service Id**	**M**	**86**	**ROE**
#2	sub-function = [eventType]	M	00-FF	LEV_ ETP
#3	eventWindowTime	M	00-FF	EWT
#4 : #(m-1)+4	eventTypeRecord[] = [eventTypeParameter 1 : eventTypeParameter m]	C_1 : C_1	00-FF : 00-FF	ETR_ ETP1 : ETPm
#n-(r-1)-1 #n-(r-1) : #n	serviceToRespondToRecord[] = [serviceId serviceParameter 1 : serviceParameter r]	C_2 C_3 : C_3	00-FF 00-FF : 00-FF	STRTR_ SI SP1 : SPr

C_1: present if the eventType requires additional parameters to be specified for the event to respond to.

C_2: mandatory to be present if the sub-function parameter is not equal to reportActivatedEvents, stopResponseOnEvent, startResponseOnEvent, ClearResponseOnEvent

C_3: present if the service request of the service to respond to requires additional service parameters

8.9.2.1 Request message sub-function Parameter $Level (LEV_) Definition

The sub-function parameter eventType is used by the ResponseOnEvent request message to specify the event to be configured in the server and to control the ResponseOnEvent set up. Each sub-function parameter value given in the table below also specifies the length of the applicable eventTypeRecord (responseRequiredIndicationBit (bit 7) not shown in table below).

Bit 6 of the eventType sub-function parameter is used to indicate whether the event shall be stored in non-volatile memory in the server and re-activated upon the next power-up of the server or if it shall terminate once the server powers down (storageState parameter):

Table 91 — eventType sub-function bit 6 definition - storageState

Bit 6 value	Description	Cvt	Mnemonic
0	**doNotStoreEvent** This value indicates that the event shall terminate when the server powers down and the server shall not continue a ResponseOnEvent diagnostic service after a reset or power on (i.e. the ResponseOnEvent service is terminated).	M	DNSE
1	**storeEvent** This value indicates that the event shall resume sending serviceToRespondTo-responses according to the ResponseOnEvent-set up after a power cycle of the server.	U	SE

Table 92 — Request message sub-function Parameter Definition

Hex (bit 5-0)	Description	Cvt	Mnemonic
00	**stopResponseOnEvent** This value is used to stop the server sending responses on event. The event logic that has been set up is not cleared but can be restarted with the startResponseOnEvent sub-function parameter. Length of eventTypeRecord: 0 byte	M	STPROE
01	**onDTCStatusChange** This value identifies the event as a new DTC detected matching the DTCstatusMask specified for this event. Length of eventTypeRecord: 1 byte Implementation hint: A server resident DTC count algorithm shall count the number of DTC's satisfying the client defined DTCStatusMask at a certain periodic rate (e.g. approximately 1 second). If the count is different from that which was calculated on the previous execution, the client shall generate the event that causes the execution of the serviceToRespondTo. The latest count shall then be stored as a reference for the next calculation. This eventType requires the specification of the DTCStatusMask in the request message (eventTypeParameter#1).	M	ONDTCS
02	**onTimerInterrupt** This value identifies the event as a timer interrupt, but the timer and its values are not part of the ResponseOnEvent service. This eventType requires the specification of more details in the request message (eventTypeRecord). Length of eventTypeRecord: 1 byte	M	OTI
03	**onChangeOfRecordDataIdentifier** This value identifies the event as a new internal data record identified by recordDataIdentifier. The data values are vehicle manufacturer specific. This eventType requires the specification of more details in the request message (eventTypeRecord). Length of eventTypeRecord: 3 bytes	M	OCOCID
04	**reportActivatedEvents** This value is used to indicate that in the positive response all events are reported that have been activated in the server with the ResponseOnEvent service (and are currently active). Length of eventTypeRecord: 0 bytes	U	RAE
05	**startResponseOnEvent** This value is used to indicate to the server to activate the event logic (including event window timer) that has been set up and start sending responses on event. Length of eventTypeRecord: 0 byte.	M	STRTROE
06	**clearResponseOnEvent** This value is used to clear the event logic that has been set up in the server (This also stops the server sending responses on event.) Length of eventTypeRecord: 0 byte.	M	CLRROE
07 - 1F	**reservedByDocument** This range of values is reserved by this document for future definition.	M	RBD
20 - 2F	**vehicleManufacturerSpecific** This range of values is reserved for vehicle manufacturer specific use.	U	VMS
30 - 3E	**systemSupplierSpecific** This range of values is reserved for system supplier specific use.	U	SSS

Table 92 — Request message sub-function Parameter Definition

Hex (bit 5-0)	Description	Cvt	Mnemonic
3F	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

Note: For easier description the request message sub-function parameters can be devided into two different groups:

– sub-function parameters to request a set up of response on event ("ROE set up sub-functions")

– sub-function parameters to control the response on event set up, like startResponseOnEvent, stopResponseOnEvent clearResponseOnEvent reportActivatedEvents. ("ROE control sub-functions")

8.9.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 93 — Request message data parameter definition

Definition
eventWindowTime The parameter eventWindowTime is used to specify a window for the event logic to be active in the server. If the parameter value of eventWindowTime is set to 02 hex then the response time is infinite. In case of an infinite event window it is recommended to close the event window by a certain signal (e.g. power off). See annex B.2 for specified eventWindowTimes. NOTE This parameter is not applicable to be evaluated by the server in case the eventType is equal to a ROE control sub-function.
eventTypeRecord This parameter record contains additional parameters for the specified eventType.
serviceToRespondToRecord This parameter record contains the service parameters (service Id and service parameters) of the service to be executed in the server each time the specified event defined in the eventTypeRecord occurs.

8.9.3 Positive response message definition

Table 94 — Positive response message definition for all sub-functions but reportActivatedEvents

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ResponseOnEvent Response Service Id**	**S**	**C6**	**ROEPR**
#2	eventType	M	00-FF	ETP
#3	numberOfIdentifiedEvents	M	00-FF	NOIE
#4	eventWindowTime	M	00-FF	EWT
#5 : #(m-1)+5	eventTypeRecord[] = [eventTypeParameter 1 : eventTypeParameter m]	 C_1 : C_1	 00-FF : 00-FF	ETR_ ETP1 : ETPm
#n-(r-1)-1 #n-(r-1) : #n	serviceToRespondToRecord[] = [serviceId serviceParameter 1 : serviceParameter r]	 M C_2 : C_2	 00-FF 00-FF : 00-FF	STRTR_ SI SP1 : SPr
C_1: present if the eventType required additional parameters to be specified for the event to respond to.				
C_2: present if the service request of the service to respond to required additional service parameters				

Table 95 — Positive response message definition, sub-function = reportActivatedEvents

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ResponseOnEvent Response Service Id**	**S**	**C6**	**ROEPR**
#2	eventType = reportActivatedEvents	M	04	ETP_RAE
#3	numberOfActivatedEvents	M	00-FF	NOIE
#4	eventTypeOfActiveEvent #1	C_1	00-FF	EVOAE
#5	eventWindowTime #1	C_1	00-FF	EWT
#6 : #(m-1)+6	eventTypeRecord #1[] = [eventTypeParameter 1 : eventTypeParameter m]	C_2 : C_2	00-FF : 00-FF	ETR_ ETP1 : ETPm
#p-(o-1)-1 #p-(o-1) : #p	serviceToRespondToRecord #1[] = [serviceId serviceParameter 1 : serviceParameter o]	C_3 C_4 : C_4	00-FF 00-FF : 00-FF	STRTR_ SI SP1 : SPo
:	:	:	:	:
#4	eventTypeOfActiveEvent #k	C_1	00-FF	EVOAE
#5	eventWindowTime #k	C_1	00-FF	EWT
#6 : #(q-1)+6	eventTypeRecord #k[] = [eventTypeParameter 1 : eventTypeParameter q]	C_2 : C_2	00-FF : 00-FF	ETR_ ETP1 : ETPm
#n-(r-1)-1 #n-(r-1) : #n	serviceToRespondToRecord #k[] = [serviceId serviceParameter 1 : serviceParameter r]	C_3 C_4 : C_4	00-FF 00-FF : 00-FF	STRTR_ SI SP1 : SPr

C_1: present if an active event is reported.

C_2: present if the reported eventType of the active event (eventTypeOfActiveEvent) requires additional parameters to be specified for the event to respond to.

C_3: mandatory to be present when reporting an active event.

C_4: present if the reported service request of the service to respond to requires additional service parameters.

8.9.3.1 Positive response message data parameter definition

Table 96 — Response message data parameter definition

Definition
eventType This parameter is an echo of the sub-function parameter of the request message.
eventTypeOfActiveEvent This parameter is an echo of the sub-function parameter of the request message that was issued to set-up the active event. The applicable values are the ones specified for the eventType sub-function parameter.

Table 96 — Response message data parameter definition

Definition
numberOfActivatedEvents This parameter contains the number of active events when the client requests to report the number of active events. This number reflects the number of events reported in the response message.
numberOfIdentifiedEvents This parameter contains the number of identified events during an active event window and is only applicable for the response message send at the end of the event window (in case of a finite event window). The initial response to the request message shall contain a zero (0) in this parameter.
eventWindowTime This parameter is an echo of the eventWindowTime parameter from the request message. When reporting an active event then this parameter contains the time remaining for the event to be active.
eventTypeRecord This parameter is an echo of the eventTypeRecord parameter from the request message. When reporting an active event then this parameter is an echo of the eventTypeRecord of the request that was issued to set-up the active event.
serviceToRespondToRecord This parameter is an echo of the serviceToRespondToRecord parameter from the request message. When reporting an active event then this parameter is an echo of the serviceToRespondToRecord of the request that was issued to set-up the active event.

8.9.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 97 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** Send if the sub-function parameter in the request message is not supported	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** Used when the server is in a critical normal mode activity and therefore cannot perform the requested functionality.	U	CNC
31	**requestOutOfRange** The server shall use this response code 1. if it detects an error in the eventTypeRecord parameter. 2. if the specified eventWindowTime is invalid	M	ROOR

8.9.5 Message flow example(s) ResponseOnEvent

For the message flow examples it is assumed, that the eventWindowTime equal to 08 hex defines an event window of 80 seconds (eventWindowTime * 10 seconds). The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

NOTE The definition of the eventWindowTime is vehicle manufacturer specific, except for certain values as specified in annex B.2.

The following conditions apply to the shown message flow examples and flowcharts:

— **Trigger signal:**
It is up to the vehicle manufacturer to define a specific trigger signal, which causes the client (external test equipment, OBD-Unit, diagnostic master, etc.) to start the ResponseOnEvent request message. This trigger signal could be enabled by an event as well as by a fixed timing schedule like a heartbeat-time (which should be greater than the eventWindowTime). Furthermore there could be a synchronous message (e.g. SYNCH-signal) on the data link used as trigger signal.

— **Open event window:**
Receiving the ResponseOnEvent request message, the server shall evalaute the request. If the evaluation was positive, the server shall set up the event logic and has to send the initial positive response message of the ResponseOnEvent service. To activate the event logic the client has to request ResponseOnEvent sub-function startResponseOnEvent. After the positive response the event logic is activated and the event window timer is running. It is up to the vehicle manufacturer to define the event window in detail, using the parameter eventWindowTime (e.g. timing window, ignition on/off window). In case of detecting the specified eventType (EART_) the server has to respond immediately with the response message corresponding to the serviceToRespondToRecord in the ResponseOnEvent request message.

— **Close_event_window:**
It is recommended to close the event window of the server according to the parameter eventWindowTime. After this action, the server has to stop sending event driven diagnostic response messages. The same could either be reached by sending the ResponseOnEvent (ROE_) request message including the parameter stopResponseOnEvent or by power off.

8.9.5.1 Example #1 - ResponseOnEvent (finite event window)

Table 98 — Set up ResponseOnEvent request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ResponseOnEvent request SID	86	ROE
#2	eventTypeRecord[eventType] = onDTCStatusChange, storageState = doNotStoreEvent responseRequired = yes	01	ET_ODTCSC
#3	eventWindowTime = 80 seconds	08	EWT
#4	eventTypeRecord[eventTypeParameter] = testFailed status	01	ETP1
#5	serviceToRespondToRecord[serviceId] = ReadDTCInformation	19	RDTCI
#6	serviceToRespondToRecord[sub-function] = reportNumberOfDTC	02	RNDTC
#7	serviceToRespondToRecord[DTCStatusMask] = testFailed status	01	DTCSM

Table 99 — ResponseOnEvent initial positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ResponseOnEvent response SID	C6	ROEPR
#2	eventType = onDTCStatusChange	01	ET_ODTCSC
#3	numberOfIdentifiedEvents = 0	00	NOIE
#4	eventWindowTime = 80 seconds	08	EWT
#5	eventTypeRecord[eventTypeParameter] = testFailed status	01	ETP1
#6	serviceToRespondToRecord[serviceId] = ReadDTCInformation	19	RDTCI
#7	serviceToRespondToRecord[sub-function] = reportNumberOfDTC	02	RNDTC
#8	serviceToRespondToRecord[DTCStatusMask] = testFailed status	01	DTCSM

The event logic is set up; now it has to be activated.

Table 100 — Start ResponseOnEvent request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ResponseOnEvent request SID	86	ROE
#2	eventTypeRecord[eventType] = startResponseOnEvent, storageState = doNotStoreEvent responseRequired = yes	05	ET_STRTROE
#3	eventWindowTime (will not be evaluated)	08	EWT

Table 101 — ResponseOnEvent positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ResponseOnEvent response SID	C6	ROEPR
#2	eventType = onDTCStatusChange	05	ET_ODTCSC
#3	numberOfIdentifiedEvents = 0	00	NOIE
#4	eventWindowTime	08	EWT

In case the specified event occurs the server sends the response message according to the specified serviceToRespondToRecord.

Table 102 — ReadDTCInformation positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDTCI
#2	DTCStatusAvailibilityMask	FF	DTCSAM
#3	DTCCount[DTCCountHighByte] = 0	00	DTCCNT_HB
#4	DTCCount[DTCCountLowByte] = 4	04	DTCCNT_LB

The message flow for the case where the client would request to report the currently active events in the server during the active event window will look as follows.

Table 103 — ResponseOnEvent request number of active events message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ResponseOnEvent request SID	86	ROE
#2	eventTypeRecord[eventType] = reportActivatedEvents, storageState = doNotStoreEvent responseRequired = yes	04	ET_RAE

Table 104 — ResponseOnEvent reportActivatedEvents positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ResponseOnEvent response SID	C6	ROEPR
#2	eventType = reportActivatedEvents	04	ET_RAE
#3	numberOfActivatedEvents = 1	01	NOAE
#4	eventTypeOfActiveEvent = onDTCStatusChange	01	ET_ODTCSC
#5	eventWindowTime = 80 seconds	08	EWT
#6	eventTypeRecord[eventTypeParameter] = testFailed status	01	ETP1
#7	serviceToRespondToRecord[serviceId] = ReadDTCInformation	19	RDTCI
#8	serviceToRespondToRecord[sub-function] = reportNumberOfDTC	02	RNDTC
#9	serviceToRespondToRecord[DTCStatusMask] = testFailed status	01	DTCSM

8.9.5.1.1 Example #1 - Flowcharts

The following flowcharts show two different kind of server behaviour:

— no event occurs within the finite event window. In this case the server has to send the response of the ResponseOnEvent at the end of the event window. It depends on the situation and the server application to send a negative response instead.

— multiple events (#1 to #n) within a finite event window. Each positive response of the serviceToRespondTo is related to an identified event (#1..#n) and shall have the same service identifier (SId) but might have different content. At the end of the event_Window the server shall transmit a positive response message of the responseOnEvent service, which indicates the numberOfIdentifiedEvents.

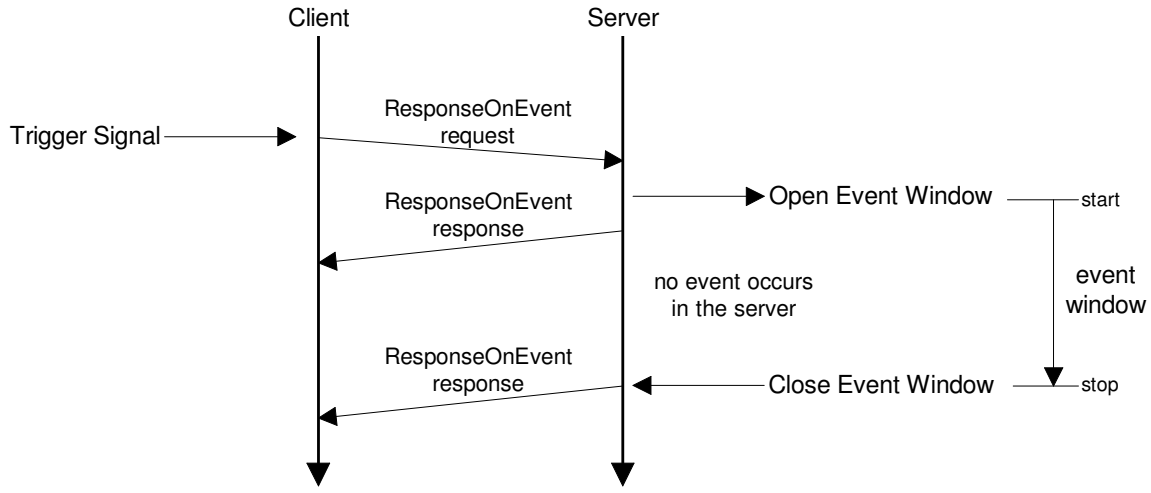


Figure 12 — Finite event window - no event during active event window

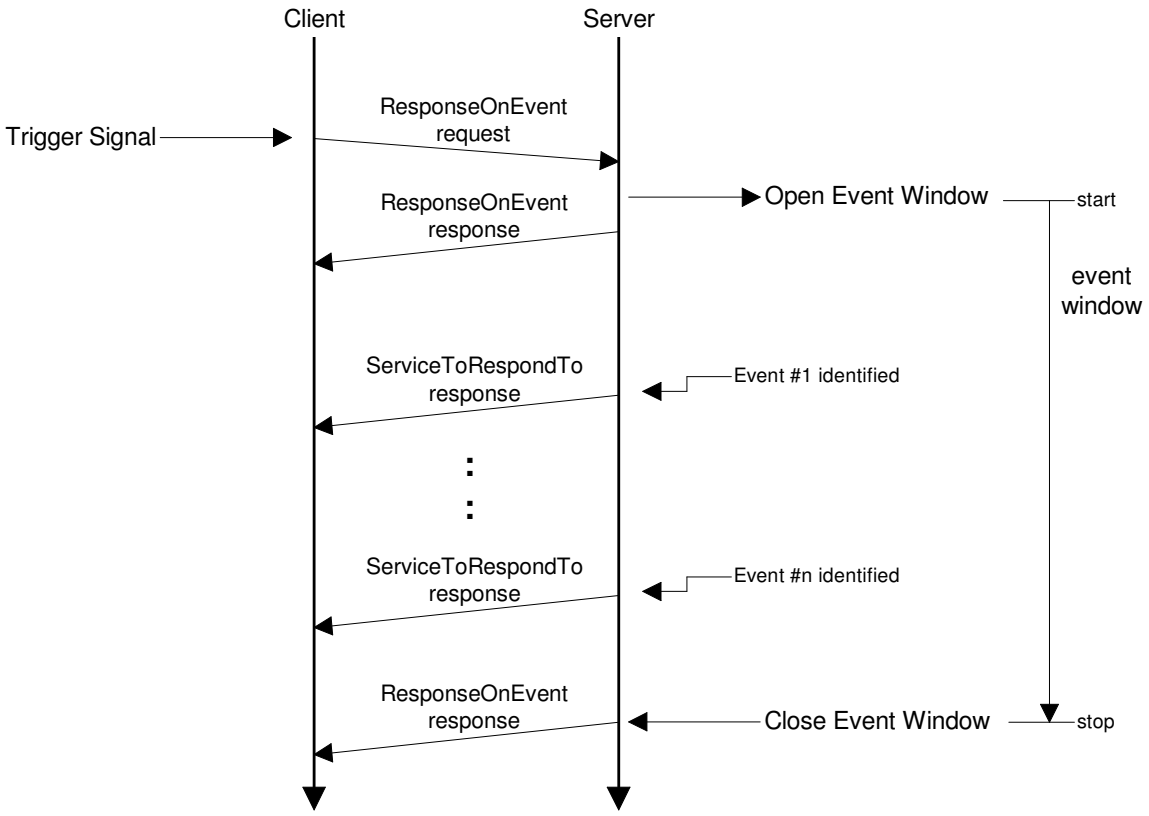


Figure 13 — Finite event window - multiple events during active event window

8.9.5.2 Example #2 - ResponseOnEvent (infinite event window)

Table 105 — ResponseOnEvent request message flow example #2

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ResponseOnEvent request SID	86	ROE
#2	eventTypeRecord[eventType] = onDTCStatusChange, storageState = doNotStoreEvent responseRequired = yes	01	ET_ODTCSC
#3	eventWindowTime = infinite	FF	EWT
#4	eventTypeRecord[eventTypeParameter] = testFailed status	01	ETP1
#5	serviceToRespondToRecord[serviceId] = ReadDTCInformation	19	RDTCI
#6	serviceToRespondToRecord[sub-function] = reportNumberOfDTC	02	RNDTC
#7	serviceToRespondToRecord[DTCStatusMask] = testFailed status	01	DTCSM

Table 106 — ResponseOnEvent initial positive response message flow example #2

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ResponseOnEvent response SID	C6	ROEPR
#2	eventType = onDTCStatusChange	01	ET_ODTCSC
#3	numberOfIdentifiedEvents = 0	00	NOIE
#4	eventWindowTime = infinite	02	EWT
#5	eventTypeRecord[eventTypeParameter] = testFailed status	01	ETP1
#6	serviceToRespondToRecord[serviceId] = ReadDTCInformation	19	RDTCI
#7	serviceToRespondToRecord[sub-function] = reportNumberOfDTC	02	RNDTC
#8	serviceToRespondToRecord[DTCStatusMask] = testFailed status	01	DTCSM

The event logic is set up; now it has to be activated.

Table 107 — Start ResponseOnEvent request message flow example #2

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ResponseOnEvent request SID	86	ROE
#2	eventTypeRecord[eventType] = startResponseOnEvent, storageState = doNotStoreEvent responseRequired = yes	05	ET_STRTROE
#3	eventWindowTime (will not be evaluated)	02	EWT

Table 108 — ResponseOnEvent positive response message flow example #2

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ResponseOnEvent response SID	C6	ROEPR
#2	eventType = onDTCStatusChange	05	ET_ODTCSC
#3	numberOfIdentifiedEvents = 0	00	NOIE
#4	eventWindowTime	02	EWT

In case the specified event occurs the server sends the response message according to the specified serviceToRespondToRecord.

Table 109 — ReadDTCInformation positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDTCI
#2	DTCStatusAvailibilityMask	xx	DTCSAM
#3	DTCCount[DTCCountHighByte] = 0	00	DTCCNT_HB
#4	DTCCount[DTCCountLowByte] = 4	04	DTCCNT_LB

8.9.5.2.1 Example #2 - Flowcharts

The following flowcharts show two different kind of server behaviour:

— no event occurs within the infinite event window.

— multiple events (#1 to #n) within a infinite event window. Each positive response of the serviceToRespondTo is related to an identified event (#1..#n) and shall have the same service identifier (SId) but might have different content.

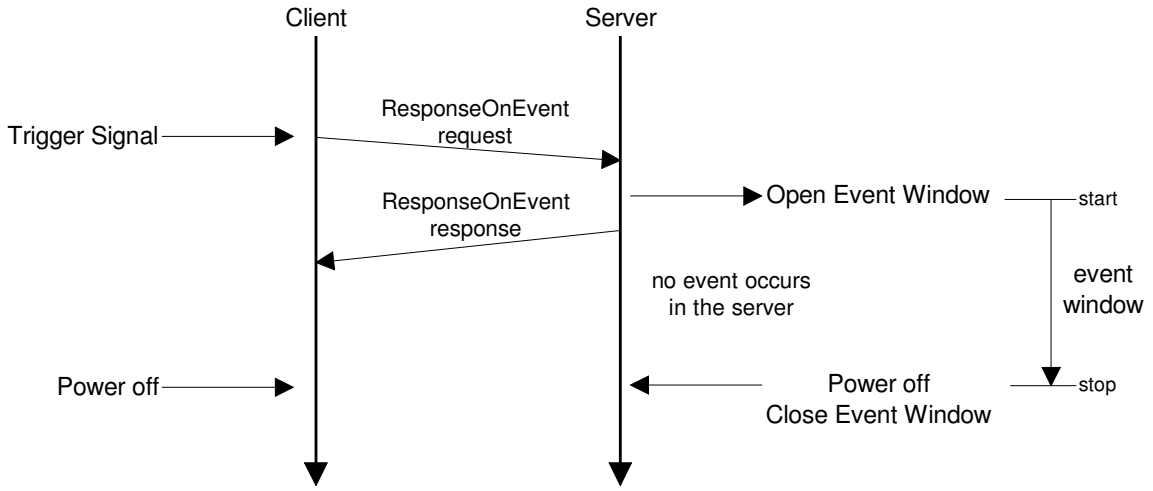


Figure 14 — Infinite event window - no event during active event window

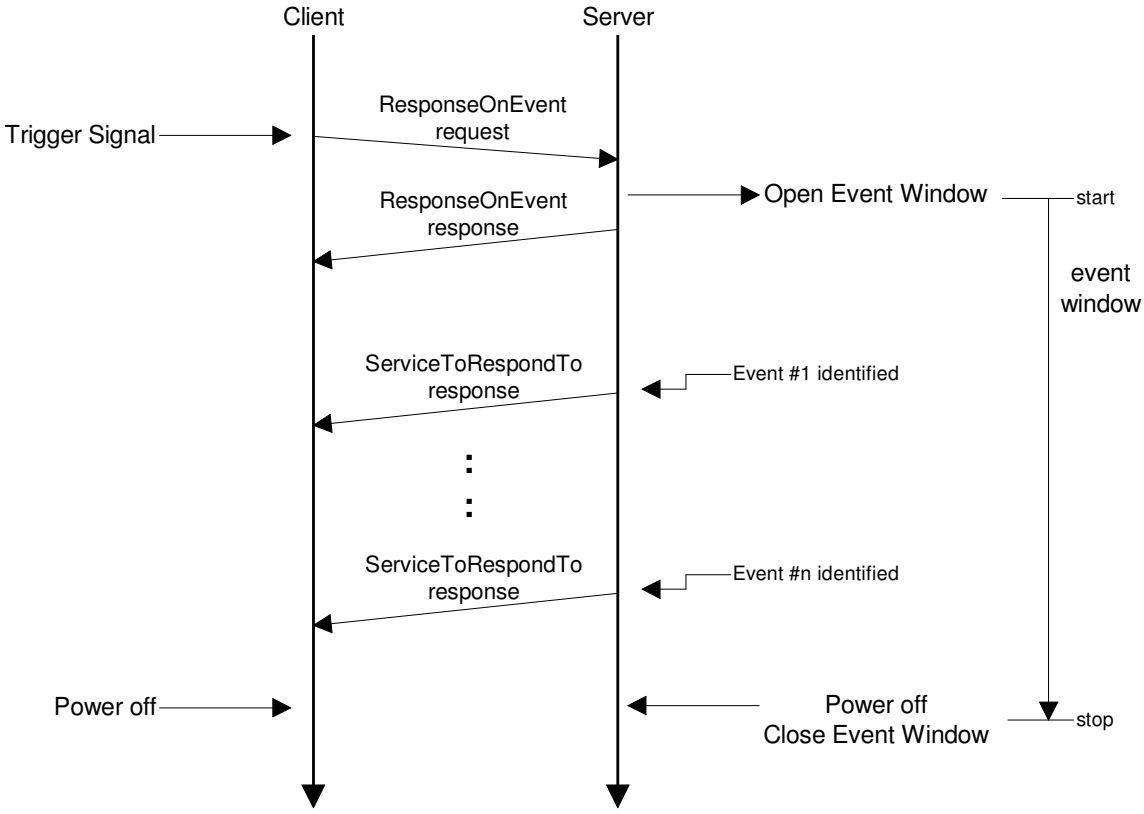


Figure 15 — Infinite event window - multiple events during active event window

8.10 LinkControl (87 hex) service

The LinkControl service is used to control the communication link baudrate between the client and the server(s) for the exchange of diagnostic data. This service optionally applies to those data link layers, which allow for a baudrate transition during an active diagnostic session.

NOTE Further details on the appliance and usage of this service on a certain data link layer can be found in the data link layer specific diagnostic services implementation specification.

8.10.1 Service description

This service is used to transition the baudrate of the data link layer. To overcome functional communication, where the baudrate has to be transitioned in multiple servers at the same time, the baudrate transition is split into two steps:

— **Step #1**: The client verifies if the transition can be performed and informs the server(s) about the baudrate to be used. Each server has to respond positively (responseRequired = yes) before the client performs step #2. This step actually does not perform the baudrate transition.

— **Step #2**: The client actually requests the transition of the baudrate. This step shall only be performed if it is verified that the baudrate transition can be performed (step #1 performed). In case of functional communication it is recommended that there shall not be any response from a server when the baudrate is transitioned (responseRequired = no), because one server might already have been transitioned to the new baudrate while others still need to transmit their response message(s) (baudrate mismatch avoidance).

The linkControlType parameter in the request message in conjunction with the conditional baudrateIdentifier/linkBaudrateRecord parameter provides a mechanism to transition to a pre-defined or specifically defined baudrate.

Any baudrate transition shall occur as follows:

— responseRequiredIndicationBit = **no**: after the successful transmission/reception of the client request message, which requests the baudrate transition.

— responseRequiredIndicationBit = **yes**: after the successful transmission/reception of the server positive response message, which confirms the successful reception of the request, which requests the baudrate transition.

NOTE This service is tied to a non-defaultSession. A session layer timer timeout will transition the server(s) back to its (their) normal speed of operation. The same applies in case an ECUReset service (11 hex) is performed. The transition into another non-defaultSession shall not influence the baudrate.

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

8.10.2 Request message definition

Table 110 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**LinkControl Request Service Id**	**M**	**87**	**LC**
#2	sub-function = [linkControlType]	M	 00-FF	LEV_ LCTP _
#3	baudrateIdentifier	C_1	00-FF	BI_
 #4 #5 #6	linkBaudrateRecord[] = [baudrateHighByte baudrateMiddleByte baudrateLowByte]	 C_2 C_2 C_2	 00-FF 00-FF 00-FF	LBR_ BRHB BRMB BRLB
C_1: This parameter is present if the sub-function parameter indicates that a verification of a fixed baudrate is done.				
C_2: This parameter is present if the sub-function parameter indicates that a verification of a specific baudrate is done.				

8.10.2.1 Request message sub-function parameter $Level (LEV_) definition

The sub-function parameter linkControlType is used by the LinkControl request message to describe the action to be performed in the server (responseRequiredIndicationBit (bit 7) not shown in table below).

Table 111 — Request message sub-function Parameter Definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**reservedByDocument** This value is reserved by this document.	M	RBD
01	**verifyBaudrateTransitionWithFixedBaudrate** This parameter is used to verify if a transition to a pre-defined baudrate, which is specified by the baudrateIdentifier data parameter can be performed.	U	VBTWFBR
02	**verifyBaudrateTransitionWithSpecificBaudrate** This parameter is used to verify if a transition to a specifically defined baudrate, which is specified by the linkBaudrateRecord data parameter can be performed.	U	VBTWSBR
03	**transitionBaudrate** This sub-function parameter requests the server(s) to transition the baudrate to the one that was specified in the preceding verification message.	U	TB
04 - 3F	**reservedByDocument** This range of values is reserved by this document for future definition.	M	RBD
40 - 5F	**vehicleManufacturerSpecific** This range of values is reserved for vehicle manufacturer specific use.	U	VMS
60 - 7E	**systemSupplierSpecific** This range of values is reserved for system supplier specific use.	U	SSS
7F	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

8.10.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 112 — Request message data parameter definition

Definition
baudrateIdentifier This conditional parameter references a fixed defined baudrate to transition to (see annex B.3).
linkBaudrateRecord This conditional parameter record contains a specific baudrate ([bit/s]) in case the sub-function parameter indicates that a specific baudrate is used.

8.10.3 Positive response message definition

Table 113 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**LinkControl Response Service Id**	**S**	**C7**	**LCPR**
#2	linkControlType	M	00-FF	LCTP

8.10.3.1 Positive response message data parameter definition

Table 114 — Response message data parameter definition

Definition
linkControlType This parameter is an echo of the linkControlType sub-function parameter from the request message.

8.10.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 115 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** Send if the sub-function parameter in the request message is not supported.	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This code shall be returned if the criteria for the requested LinkControl are not met.	M	CNC
24	**requestSequenceError** This code shall be returned it the client requests the transition of the baudrate without a preceding verification step, which specifies the baudrate to transition to.	M	RSE
31	**requestOutOfRange** This code shall be returned if 1. the requested fixed baudrate (baudrateIdentifier) is invalid 2. the specific baudrate (linkBaudrateRecord) is invalid.	M	ROOR

8.10.5 Message flow example(s) LinkControl

8.10.5.1 Example #1 - Transition baudrate to fixed baudrate (PC baudrate 115200 kBit/s)

8.10.5.1.1 Step#1: Verify if all criteria are met for a baudrate switch

Table 116 — LinkControl request message flow example #1 - step #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	LinkControl request SID	87	LC
#2	linkControlType = verifyBaudrateTransitionWithFixedBaudrate, responseRequired = yes	01	VBTWFBR
#3	baudrateIdentifier = PC115200Baud	05	BI_PC115200

Table 117 — LinkControl positive response message flow example #1 - step #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	LinkControl response SID	C7	LCPR
#2	linkControlType = verifyBaudrateTransitionWithFixedBaudrate	01	VBTWFBR

8.10.5.1.2 Step#2: Transition the baudrate

Table 118 — LinkControl request message flow example #1 - step #2

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	LinkControl request SID	87	LC
#2	linkControlType = transitionBaudrate, responseRequired = no	83	TB

There is no response from the server(s). The client and the server(s) have to transition the baudrate of their communication link.

8.10.5.2 Example #2 - Transition baudrate to specific baudrate (150kBit/s)

8.10.5.2.1 Step#1: Verify if all criteria are met for a baudrate switch

Table 119 — LinkControl request message flow example #2 - step #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	LinkControl request SID	87	LC
#2	linkControlType = verifyBaudrateTransitionWithSpecificBaudrate, responseRequired = yes	02	VBTWSBR
#3	linkBaudrateRecord[baudrateHighByte] (150kBit/s)	02	BR_BRHB
#4	linkBaudrateRecord[baudrateMiddleByte]	49	BR_BRMB
#5	linkBaudrateRecord[baudrateLowByte]	F0	BR_BRLB

Table 120 — LinkControl positive response message flow example #2 - step #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	LinkControl response SID	C7	LCPR
#2	linkControlType = verifyBaudrateTransitionWithSpecificBaudrate	02	VBTWSBR

8.10.5.2.2 Step#2: Transition the baudrate

Table 121 — LinkControl request message flow example #2 - step #2

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	LinkControl request SID	87	LC
#2	linkControlType = transitionBaudrate, responseRequired = no	83	TB

There is no response from the server(s). The client and the server(s) have to transition the baudrate of their communication link.

9 Data Transmission functional unit

Table 122 —Data Transmission functional unit

Service	Description
ReadDataByIdentifier	The client requests to read the current value of a record identified by a provided recordDataIdentifier.
ReadMemoryByAddress	The client requests to read the current value of the provided memory range.
ReadScalingDataByIdentifier	The client requests to read the scaling information of a record identified by a provided recordDataIdentifier.
ReadDataByPeriodicIdentifier	The client requests to schedule data in the server for periodic transmission.
DynamicallyDefineDataIdentifier	The client requests to dynamically define data Identifiers that may subsequently be read by the readDataByIdentifier service.
WriteDataByIdentifier	The client requests to write a record specified by a provided recordDataIdentifier.
WriteMemoryByAddress	The client requests to overwrite a provided memory range.

9.1 ReadDataByIdentifier (22 hex) service

The ReadDataByIdentifier service allows the client to request data record values from the server identified by one or more recordDataIdentifiers.

9.1.1 Service description

The client request message contains one or more two (2) byte recordDataIdentifier values that identify data record(s) maintained by the server (refer to annex C.1 for allowed recordDataIdentifier values). The format and definition of the dataRecord shall be vehicle manufacturer or system supplier specific, and may include analog input and output signals, digital input and output signals, internal data, and system status information if supported by the server.

Upon receiving a ReadDataByIdentifier request, the server shall access the data elements of the records specified by the recordDataIdentifier parameter(s) and transmit their value in one single ReadDataByIdentifier positive response containing the associated dataRecord parameter(s).

The server shall behave as folllows after the reception of a request message that contains one or more recordIdentifiers, which are not supported by the server (compliant with ISO 15031-5):

— Physical communication:

  — If a single requested recordDataIdentifier is not supported by the server then a negative response message with response code $31 shall be sent.

  — In case the client requests multiple recordDataIdentifiers where at least one recordDataIdentifier is not supported then the server shall send a negative response message with response code $31. There shall be no positive response for the supported recordDataIdentifiers.

— Functional communication:

  — If a single requested recordDataIdentifier is not supported by the server then no response message shall be sent.

  — In case the client requests multiple recordDataIdentifiers where at least one recordDataIdentifier is supported then the server shall send a positive response message with the supported recordDataIdentifiers and the associated record data.

The server may limit the number of recordDataIdentifiers that can be simultaneously supported as agreed upon by the vehicle manufacturer and system supplier. Exceeding the maximum number of recordDataIdentifiers that can be simultaneously supported shall result in a negative response with response code 31 hex.

9.1.2 Request message definition

Table 123 — Request message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDataByIdentifier Request Service Id**	**M**	**22**	**RDBI**
	recordDataIdentifier[] #1 = [			RDI_
#2	byte 1 (MSB)	M	00-FF	B1
#3	byte 2]	M	00-FF	B2
:	:	:	:	:
	recordDataIdentifier[] #m = [			RDI_
#n-1	byte 1 (MSB)	U	00-FF	B1
#n	byte 2]	U	00-FF	B2

9.1.2.1 Request message sub-function parameter $Level (LEV_) Definition

This service does not use a sub-function parameter.

9.1.2.2 Request message data parameter definition

The following data-parameters are defined for this service.

Table 124 — Request message data parameter definition

Definition
recordDataIdentifier (#1 to #m) This parameter identifies the server data record(s) that are being requested by the client (see annex C.1 for detailed parameter definition).

9.1.3 Positive response message definition

Table 125 — Positive response message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDataByIdentifier Response Service Id**	**M**	**62**	**RDBIPR**
	recordDataIdentifier[] #1 = [			RDI_
#2	byte 1 (MSB)	M	00-FF	B1
#3	byte 2]	M	00-FF	B2
	dataRecord[] #1 = [			DREC_
#4	data#1	M	00-FF	DATA_1
:	:	:	:	:
#(k-1)+4	data#k]	U	00-FF	DATA_m
:	:	:	:	:
	recordDataIdentifier[] #m = [			RDI_
#n-(o-1)-2	byte 1 (MSB)	U	00-FF	B1
#n-(o-1)-1	byte 2]	U	00-FF	B2
	dataRecord[] #m = [			DREC_
#n-(o-1)	data#1	U	00-FF	DATA_1
:	:	:	:	:
#n	data#o]	U	00-FF	DATA_k

9.1.3.1 Positive response message data parameter definition

Table 126 — Response message data parameter definition

Definition
recordDataIdentifier (#1 to #m)
This parameter is an echo of the data parameter recordDataIdentifier from the request message.
dataRecord (#1 to #k/o)
This parameter is used by the ReadDataByIdentifier positive response message to provide the requested data record values to the client. The content of the dataRecord is not defined in this document and is vehicle manufacturer specific.

9.1.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 127 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** This response code shall be sent if the length of the request message is invalid.	M	IMLOIF
22	**conditionsNotCorrect** This response code shall be sent if the operating conditions of the server are not met to perform the required action.	U	CNC
31	**requestOutOfRange** This code shall be sent if 1. one or multiple of the requested recordDataIdentifier value(s) is (are) not supported by the device (physical addressing only). 2. the client exceeded the maximum number of recordDataIdentifiers allowed to be requested at a time.	M	ROOR

Table 127 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
33	**securityAccessDenied** This code shall be sent if the recordDataIdentifier is secured and the server is not in an unlocked state.	M	SAD

9.1.5 Message flow example ReadDataByIdentifier

This section specifies the conditions to be fulfilled for the example to perform a ReadDataByIdentifier service. The client may request a recordDataIdentifier data at any time independent of the status of the server.

The recordDataIdentifier examples below are specific to a powertrain device (e.g., engine control module). Refer to ISO/DIS 15031-2 for further details regarding accepted terms/definitions/acronyms for emission related systems.

The first example reads a single recordDataIdentifier containing a single piece of information (where recordDataIdentifier F190 hex contains the VIN number).

The second example demonstrates requesting of multiple recordDataIdentifiers with a single request (where recordDataIdentifier 010A hex contains engine coolant temperature, throttle position, engine speed, manifold absolute pressure, mass air flow, vehicle speed sensor, barometric pressure, calculated load value, idle air control, and accelerator pedal position, and recordDataIdentifier 0110 hex contains battery positive voltage).

9.1.5.1 Example #1: read single recordDataIdentifier F190 hex (VIN number)

Table 128 — ReadDataByIdentifier request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier request SID	22	RDBI
#2	recordDataIdentifier [byte 1] (MSB)	F1	RDI_B1
#3	recordDataIdentifier [byte 2]	90	RDI_B2

Table 129 — ReadDataByIdentifier positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier response SID	62	RDBIPR
#2	recordDataIdentifier [byte 1] (MSB)	F1	RDI_B1
#3	recordDataIdentifier [byte 2]	90	RDI_B2
#4	dataRecord [data_1] = VIN Digit 1 = “W”	57	DREC_DATA1
#5	dataRecord [data_2] = VIN Digit 2 = “0”	30	DREC_DATA2
#6	dataRecord [data_3] = VIN Digit 3 = “L”	4C	DREC_DATA3
#7	dataRecord [data_4] = VIN Digit 4 = “0”	30	DREC_DATA4
#8	dataRecord [data_5] = VIN Digit 5 = “0”	30	DREC_DATA5
#9	dataRecord [data_6] = VIN Digit 6 = “0”	30	DREC_DATA6
#10	dataRecord [data_7] = VIN Digit 7 = “0”	30	DREC_DATA7
#11	dataRecord [data_8] = VIN Digit 8 = “4”	34	DREC_DATA8
#12	dataRecord [data_9] = VIN Digit 9 = “3”	33	DREC_DATA9
#13	dataRecord [data_10] = VIN Digit 10 = “M”	4D	DREC_DATA10
#14	dataRecord [data_11] = VIN Digit 11 = “B”	42	DREC_DATA11
#15	dataRecord [data_12] = VIN Digit 12 = “5”	35	DREC_DATA12
#16	dataRecord [data_13] = VIN Digit 13 = “4”	34	DREC_DATA13
#17	dataRecord [data_14] = VIN Digit 14 = “1”	31	DREC_DATA14
#18	dataRecord [data_15] = VIN Digit 15 = “3”	33	DREC_DATA15
#19	dataRecord [data_16] = VIN Digit 16 = “2”	32	DREC_DATA16
#20	dataRecord [data_17] = VIN Digit 17 = “6”	36	DREC_DATA17

9.1.5.2 Example #2: Read multiple recordDataIdentifiers 010A hex and 0110 hex

Table 130 — ReadDataByIdentifier request message flow example #2

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier request SID	22	RDBI
#2	recordDataIdentifier #1 [byte 1] (MSB)	01	RDI_B1
#3	recordDataIdentifier #1 [byte 2]	0A	RDI_B2
#4	recordDataIdentifier #2 [byte 1] (MSB)	01	RDI_B1
#5	recordDataIdentifier #2 [byte 2]	10	RDI_B2

Table 131 — ReadDataByIdentifier positive response message flow example #2

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier response SID	62	RDBIPR
#2	recordDataIdentifier [byte 1] (MSB)	01	RDI_B1
#3	recordDataIdentifier [byte 2]	0A	RDI_B2
#4	dataRecord[data#1] = ECT	A6	DREC_DATA1
#5	dataRecord[data#2] = TP	66	DREC_DATA2
#6	dataRecord[data#3] = RPM (high byte)	07	DREC_DATA3
#7	dataRecord[data#4] = RPM (low byte)	50	DREC_DATA4
#8	dataRecord[data#5] = MAP	20	DREC_DATA5
#9	dataRecord[data#6] = MAF	1A	DREC_DATA6
#10	dataRecord[data#7] = VSS	00	DREC_DATA7
#11	dataRecord[data#8] = BARO	63	DREC_DATA8
#12	dataRecord[data#9] = LOAD	4A	DREC_DATA9
#13	dataRecord[data#10] = IAC	82	DREC_DATA10
#14	dataRecord[data#11] = APP	7E	DREC_DATA11
#15	recordDataIdentifier [byte 1] (MSB)	01	RDI_B1
#16	recordDataIdentifier [byte 2]	10	RDI_B2
#17	dataRecord[data#1] = B+	8C	DREC_DATA1

9.2 ReadMemoryByAddress (23 hex) service

The ReadMemoryByAddress service allows the client to request memory data from the server via provided starting address and size of memory to be read.

9.2.1 Service description

The ReadMemoryByAddress request message is used to request memory data from the server identified by the parameter memoryAddress and memorySize. The number of bytes used for the memoryAddress and memorySize parameter is defined by addressAndLengthFormatIdentifier (low and high nibble).

It is also possible to use a fixed addressAndLengthFormatIdentifier and unused bytes within the memoryAddress or memorySize parameter are padded with the value 00 hex in the higher range address locations.

In case of overlapping memory areas it is possible to use an additional memoryAddress byte as a memoryIdentifier. (e.g. use of internal and external flash)

The server sends data record values via the ReadMemoryByAddress positive response message. The format and definition of the dataRecord parameter shall be vehicle manufacturer specific. The dataRecord parameter may include analog input and output signals, digital input and output signals, internal data and system status information if supported by the server.

9.2.2 Request message definition

Table 132 — Request message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadMemoryByAddress Request Service Id**	**M**	**23**	**RMBA**
#2	addressAndLengthFormatIdentifier	M	00-FF	ALFID
#3 : #(m-1)+3	memoryAddress[] = [byte 1 (MSB) : byte m]	M : C_1	00-FF : 00-FF	MA_ B1 : Bm
#n-(k-1) : #n	memorySize[] = [byte 1 (MSB) : byte k]	M : C_2	00-FF : 00-FF	MS_ B1 : Bk
C_1: The presence of this parameter depends on address length information parameter of the addressAndLengthFormatIdentifier				
C_2: The presence of this parameter depends on the memory size length information of the addressAndLengthFormatIdentifier.				

9.2.2.1 Request message sub-function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

9.2.2.2 Request message data parameter definition

The following data-parameters are defined for this service.

Table 133 — Request message data parameter definition

Definition
addressAndLengthFormatIdentifier This parameter is a one byte value with each nibble encoded separately (see annex H.1 for example values): bit 7 - 4: Length (number of bytes) of the memorySize parameter bit 3 - 0: Length (number of bytes) of the memoryAddress parameter
memoryAddress The parameter memoryAddress is the starting address of server memory from which data is to be retrieved. The number of bytes used for this address is defined by the low nibble (bit 3 - 0) of the addressFormatIdentifier. Byte m in the memoryAddress parameter is always the least significant byte of the address being referenced in the server. The most significant byte of the address can be used as a memoryIdentifier. An example of the use of a memoryIdentifier would be a dual processor server with 16 bit addressing and memory address overlap (when a given address is valid for either processor but yields a different physical memory device or internal and external flash is used). In this case, an otherwise unused byte within the memoryAddress parameter can be specified as a memoryIdentifier used to select the desired memory device. Usage of this functionality shall be as defined by vehicle manufacturer / system supplier.
memorySize The parameter memorySize in the ReadMemoryByAddress request message specifies the number of bytes to be read starting at the address specified by memoryAddress in the server's memory. The number of bytes used for this size is defined by the high nibble (bit 7 - 4) of the addressFormatIdentifier.

9.2.3 Positive response message definition

Table 134 —Positive response message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadMemoryByAddress Response Service Id**	**M**	**63**	**RMBAPR**
#2 : #n	dataRecord[ ] = [data#1 : data#m]	M : U	00-FF : 00-FF	DREC_ DATA_1 : DATA_m

9.2.3.1 Positive response message data parameter definition

Table 135 — Response message data parameter definition

Definition
dataRecord This parameter is used by the ReadMemoryByAddress positive response message to provide the requested data record values to the client. The content of the dataRecord is not defined in this document and is vehicle manufacturer specific.

9.2.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 136 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This response code shall be sent if the operating conditions of the server are not met to perform the required action.	U	CNC
31	**requestOutOfRange** This response code shall be sent if: 1. any memory address within the interval [$MA, ($MA + $MS -$1)] is invalid, 2. any memory address within the interval [$MA, ($MA + $MS -$1)] is restricted, 3. the memorySize parameter value in the request message is greater than the maximum value supported by the server, 4. the specified addressAndLengthFormatIdentifier is not valid.	M	ROOR
33	**SecurityAccessDenied** This code shall be sent if any memory address within the interval [$MA, ($MA + $MS -$1)] is secure and the server is locked.	M	SAD

9.2.5 Message flow example ReadMemoryByAddress

This section specifies the conditions to be fulfilled for the example to perform a ReadMemoryByAddress service. The service in this example is not limited by any restriction of the server.

9.2.5.1 Example #1: ReadMemoryByAddress - 4-byte (32-bit) addressing

The client reads 259 data bytes from the server's memory starting at memory address 20481392 hex.

Table 137 — ReadMemoryByAddress request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadMemoryByAddress request SID	23	RMBA
#2	addressAndLengthFormatIdentifier	24	ALFID
#3	memoryAddress[byte 1] (MSB)	20	MA_B1
#4	memoryAddress[byte 2]	48	MA_B2
#5	memoryAddress[byte 3]	13	MA_B3
#6	memoryAddress[byte 4]	92	MA_B4
#7	memorySize [byte 1] (MSB)	01	MS_B1
#8	memorySize [byte 2]	03	MS_B2

Table 138 — ReadMemoryByAddress positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadMemoryByAddress response SID	63	RMBAPR
#2	dataRecord [data#1] (memory cell #1)	00	DREC_DATA_1
:	:	:	:
#259+1	dataRecord [data#3] (memory cell #259)	8C	DREC_DATA_259

9.2.5.2 Example #2: ReadMemoryByAddress - 2-byte (16-bit) addressing.

The client reads five data bytes from the server's memory starting at memory address 4813 hex.

Table 139 — ReadMemoryByAddress request message flow example #2

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadMemoryByAddress request SID	23	RMBA
#2	addressAndLengthFormatIdentifier	12	ALFID
#3	memoryAddress[byte 1 (MSB)]	48	MA_B1
#4	memoryAddress[byte 2 (LSB)]	13	MA_B2
#5	memorySize [byte 1]	05	MS_B1

Table 140 — ReadMemoryByAddress positive response message flow example #2

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadMemoryByAddress response SID	63	RMBAPR
#2	dataRecord [data#1] (memory cell #1)	43	DREC_DATA_1
#3	dataRecord [data#2] (memory cell #2)	2A	DREC_DATA_2
#4	dataRecord [data#3] (memory cell #3)	07	DREC_DATA_3
#5	dataRecord [data#4] (memory cell #4)	2A	DREC_DATA_4
#6	dataRecord [data#5] (memory cell #5)	55	DREC_DATA_5

9.2.5.3 Example #3: ReadMemoryByAddress, 3-byte (24-bit) addressing

The client reads three data bytes from the server's external RAM cells starting at memory address 204813 hex.

Table 141 — ReadMemoryByAddress request message flow example #3

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadMemoryByAddress request SID	23	RMBA
#2	addressAndLengthFormatIdentifier	23	ALFID
#3	memoryAddress[byte 1 (MSB)]	20	MA_B1
#4	memoryAddress[byte 2]	48	MA_B2
#5	memoryAddress[byte 3 (LSB)]	13	MA_B3
#6	memorySize [byte 1 (MSB)]	00	MS_B1
#7	memorySize [byte 2 (LSB)]	03	MS_B2

Table 142 — ReadMemoryByAddress first positive response message, example #3

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadMemoryByAddress response SID	63	RMBAPR
#2	dataRecord [data#1] (memory cell #1)	00	DREC_DATA_1
#3	dataRecord [data#2] (memory cell #2)	01	DREC_DATA_2
#4	dataRecord [data#3] (memory cell #3)	8C	DREC_DATA_3

9.3 ReadScalingDataByIdentifier (24 hex) service

The ReadScalingDataByIdentifier service allows the client to request scaling data record information from the server identified by one or more recordDataIdentifiers.

9.3.1 Service description

The client request message contains a two byte recordDataIdentifier value that identifies data record(s) maintained by the server (refer to annex C.1 for allowed recordDataIdentifier values). The format and definition of the dataRecord shall be vehicle manufacturer specific, and may include analog input and output signals, digital input and output signals, internal data, and system status information if supported by the server.

Upon receiving a ReadScalingDataByIdentifier request, the server shall access the scaling information associated with the specified recordDataIdentifier parameter and transmit the scaling information values in one ReadScalingDataByIdentifier positive response.

9.3.2 Request message definition

Table 143 — Request message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadScalingDataByIdentifier Request Service Id**	**M**	**24**	**RSDBI**
	recordDataIdentifier[] = [			RDI_
#2	byte 1 (MSB)	M	00-FF	B1
#3	byte 2]	M	00-FF	B2

9.3.2.1 Request message sub-function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

9.3.2.2 Request message data parameter definition

The following data-parameters are defined for this service.

Table 144 — Request message data parameter definition

Definition
RecordDataIdentifier
This parameter identifies the server data record that is being requested by the client (see annex C.1 for detailed parameter definition).

9.3.3 Positive response message definition

Table 145 — Positive response message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadScalingDataByIdentifier Response Service Id**	**M**	**64**	**RSDBIPR**
#2 #3	recordDataIdentifier[] = [byte 1 (MSB) byte 2]	M M	00-FF 00-FF	RDI_ B1 B2
#4	scalingByte #1	M	00-FF	SB_1
#5 : #(p-1)+5	ScalingByteExtension []#1 = [ScalingByteExtensionParameter#1 : ScalingByteExtensionParameter#p]	C_1 : C_1	00-FF : 00-FF	SBE_ PAR1 : PARp
:	:	:	:	:
#n-r	scalingByte #k	C_2	00-FF	SB_k
#n-(r-1) : #n	ScalingByteExtension [] #k = [ScalingByteExtensionParameter#1 : ScalingByteExtensionParameter#r]	C_1 : C_1	00-FF : 00-FF	SBE_ PAR1 : PARr

C_1: The presence of this parameter depends on the scalingByte high nibble. It is mandatory to be present if the scalingByte high nibble is encoded as formula, unit/format, or bitMappedReportedWithOutMask.

C_2: The presence of this parameter depends on whether the encoding of the scaling information requires more than one byte.

9.3.3.1 Positive response message data parameter definition

Table 146 — Response message data parameter definition

Definition
recordDataIdentifier This parameter is an echo of the data parameter recordDataIdentifier from the request message.
scalingByte (#1 to #k) This parameter is used by the ReadScalingDataByIdentifier positive response message to provide the requested scaling data record values to the client (see Annex C.2 for detailed parameter definition).
scalingByteExtention (#1 to #p / #1 to #r) This parameter is used to provide additional information for scalingBytes with a high nibble encoded as formula, unit/format, or bitmappedReportedWithOutMask (see Annex C.3 for detailed parameter definition).

9.3.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 147 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** This response code shall be sent if the length of the request message is invalid.	M	IMLOIF
22	**conditionsNotCorrect** This response code shall be sent if the operating conditions of the server are not met to perform the required action.	U	CNC
31	**requestOutOfRange** This return code shall be sent if: 1. the requested recordDataIdentifier value is not supported by the device (physical addressing only), 2. the requested recordDataIdentifier value is supported by the device, but no scaling information is available for the specified recordDataIdentifier.	M	ROOR
33	**securityAccessDenied** This code shall be sent if the recordDataIdentifier is secured and the server is not in an unlocked state.	M	SAD

9.3.5 Message flow example ReadScalingDataByIdentifier

This section specifies the conditions to be fulfilled for the example to perform a ReadScalingDataByIdentifier service. The client may request recordDataIdentifier scaling data at any time independent of the status of the server.

The first example reads the scaling information associated with recordDataIdentifier F190, which contains a single piece of information (17 character VIN number).

The second example demonstrates the use of a formula and unit identifier for specifiying a data variable in a server.

The third example illustrates the use of readScalingDataByIdentifier to return the supported bits (validity mask) for a bit mapped recordDataIdentifier that is reported without the mask through the use of readDataByIdenditfier.

9.3.5.1 Example #1: readScalingDataByIdentifier wth recordDataIdentifier F190 hex (VIN number)

Table 148 — ReadScalingDataByIdentifier request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadScalingDataByIdentifier request SID	24	RSDBI
#2	recordDataIdentifier [byte 1] (MSB)	F1	RDI_B1
#3	recordDataIdentifier [byte 2] (LSB)	90	RDI_B2

Table 149 — ReadScalingDataByIdentifier positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadScalingDataByIdentifier response SID	64	RSDBIPR
#2	recordDataIdentifier [byte 1] (MSB)	F1	RDI_B1
#3	recordDataIdentifier [byte 2] (LSB)	90	RDI_B2
#4	scalingByte#1 {ASCII, 15 data bytes}	6F	SB_1
#5	scalingByte#2 {ASCII, 2 data bytes}	62	SB_2

9.3.5.2 Example #2: readScalingDataByIdentifier wth recordDataIdentifier 0105 hex (Vehicle Speed)

Table 150 — ReadScalingDataByIdentifier request message flow example #2

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadScalingDataByIdentifier request SID	24	RSDBI
#2	recordDataIdentifier [byte 1] (MSB)	01	RDI_B1
#3	recordDataIdentifier [byte 2] (LSB)	05	RDI_B2

Table 151 — ReadScalingDataByIdentifier positive response message flow example #2

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadScalingDataByIdentifier response SID	64	RSDBIPR
#2	recordDataIdentifier [byte 1] (MSB)	01	RDI_B1
#3	recordDataIdentifier [byte 2] (LSB)	05	RDI_B2
#4	scalingByte#1 {unsigned numeric, 1 data byte}	01	SBYT_1
#5	scalingByte#2 {formula, 0 data bytes}	90	SB_2
#6	scalingByteExtention #2 [byte 1] {formulaIdentifier = C0 * x + C1}	00	SBE_21
#7	scalingByteExtention #2 [byte 2] {C0 high byte}	A0	SBE_22
#8	scalingByteExtention #2 [byte 3] {C0 low byte} [C0 = 75 * 10^{-2}]	4B	SBE_23
#9	scalingByteExtention #2 [byte 4] {C1 high byte}	00	SBE_24
#10	scalingByteExtention #2 [byte 5] {C1 low byte} [C1 = 30 * 10^{0}]	1E	SBE_25
#11	scalingByte#3 {unit / format, 0 data bytes}	A0	SB_3
#12	scalingByteExtention #3 [byte 1] {unit ID, km/h}	30	SBE_31

Using the information contained in Annex C.2 for decoding the scalingBytes, constants (C0, C1), and units, the data variable of vehicle speed is calculated using the following formula:

Vehicle Speed = (0.75 * x + 30) km/h

where 'x' is the actual data stored in the server and is identified by recordDataIdentifier 0105 hex.

9.3.5.3 Example #3: readScalingDataByIdentifier wth recordDataIdentifier 0967 hex

This example shows how a client could determine which bits are supported for a recordDataIdentifier in a server that is formatted as a bit mapped record reported without a validity mask.

The example recordDataIdentifier (0967 hex) is defined in the table below.

Table 152 — Example Data Definition

Data Byte	Bit(s)	Description
#1	7-4	Unusued
	3	Medium speed fan is commanded on
	2	Medium speed fan output fault detected
	1	Purge monitor soak time status flag
	0	Purge monitor idle test is prevented due to refuel event
#2	7	Check fuel cap light is commanded on
	6	Check fuel cap light output fault detected
	5	Fan control A output fault detected
	4	Fan control B output fault detected
	3	High speed fan output fault detected
	2	High speed fan output is commanded on
	1	Purge monitor idle test (small leak) ready to run
	0	Purge monitor small leak has been monitored

Table 153 — ReadScalingDataByIdentifier request message flow example #3

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadScalingDataByIdentifier request SID	24	RSDBI
#2	recordDataIdentifier [byte 1] (MSB)	09	RDI_B1
#3	recordDataIdentifier [byte 2] (LSB)	67	RDI_B2

Table 154 — ReadScalingDataByIdentifier positive response message flow example #3

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadScalingDataByIdentifier response SID	64	RSDBIPR
#2	recordDataIdentifier [byte 1] (MSB)	09	RDI_B1
#3	recordDataIdentifier [byte 2] (LSB)	67	RDI_B2
#4	scalingByte#1 {bitMappedReportedWithOutMask, 2 data bytes}	22	SBYT_1
#5	scalingByteExtention #1 [byte 1] {dataRecord#1 Validity Mask}	03	SBYE_11
#6	scalingByteExtention #1 [byte 2] {dataRecord#2 Validity Mask}	43	SBYE_12

The above example makes the assumption that the only bits supported (i.e., that contain information) for this dataRecordIdentifier in the server are byte 1, bits 1 and 0, and byte 2, bits 6, 1, and 0.

9.4 ReadDataByPeriodicIdentifier (2A hex) service

The ReadDataByPeriodicIdentifier service allows the client to request the periodic transmission of data record values from the server identified by one or more periodicRecordDataIdentifiers.

9.4.1 Service description

The client request message contains one or more 1-byte periodicRecordDataIdentifier values that identify data record(s) maintained by the server. The periodicRecordDataIdentifier represents the low byte of a recordDataIdentifier out of the 2-byte recordDataIdentifier range reserved for this service (F2xx hex, refer to annex C.1 for allowed periodicRecordDataIdentifier values), e.g. the periodicRecordDataIdentifier E3 hex used in this service is the recordDataIdentifier F2E3 hex.

The format and definition of the dataRecord shall be vehicle manufacturer specific, and may include analog input and output signals, digital input and output signals, internal data, and system status information if supported by the server.

Upon receiving a ReadDataByPeriodicIdentifier request other than stopSending the server shall check whether the conditions are correct to execute the service. If the conditions are correct then the server shall transmit a positive response message, including only the service identifier. The server shall never transmit a negative response message once it has accepted the initial request message by responding positively.

Following the initial positive response message the server shall access the data elements of the records specified by the periodicRecordDataIdentifier parameter(s) and transmit their value in separate ReadDataByPeriodicIdentifier positive response messages for each periodicRecordDataIdentifier containing the associated dataRecord parameters.

There are two types of periodic data response messages defined to transmit the periodicRecordDataIdentifier data to the client following the initial positive response message. Those are defined in order to maximize the useable data portion as provided by certain data link layers:

— Response message type #1: Including the service identifier, the echo of the periodicRecordDataIdentifier and the data of the periodicRecordDataIdentifier.

— Response message type #2: Including the periodicRecordDataIdentifier and the data of the periodicRecordDataIdentifier.

The mapping of the response message types onto a certain data link layers is described in the appropriate implementation specificiations of ISO 14229-1.

The periodic rate is defined as the time between any two consecutive response messages of the same periodicRecordDataIdentifier when it is scheduled by this service (see section 9.4.5.2 for examples). The specific values that apply to the defined periodic rates (transmissionMode parameter) and their tolerances are vehicle manufacturer specific.

Upon receiveing a ReadDataByPeriodicIdentifier request including the transmissionMode stopSending the server shall either stop the periodic transmission of the periodicRecordDataIdentifier(s) contained in the request message or stop the transmission of all periodicRecordDataIdentifier if no specific one is specified in the request message. The response message to this transmissionMode only contains the service identifier.

The server may limit the number of periodicRecordDataIdentifiers that can be simultaneously supported as agreed upon by the vehicle manufacturer and system supplier. Exceeding the maximum number of periodicRecordDataIdentifier that can be simultaneously supported shall result in a single negative response and none of the periodicRecordDataIdentifier shall be scheduled. Repetition of the same periodicRecordDataIdentifier in a single request message is not allowed, and shall also result in a negative response.

The client can either stop the transmission of one or multiple individual periodicRecordDataIdentifiers or it can stop the transmission of all scheduled periodicRecordDataIdentifiers. In both cases the server shall only send

a single positive response message indicating that it has stopped the scheduling of the periodicRecordDataIdentifiers given in the request message.

9.4.2 Request message definition

Table 155 — Request message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDataByPeriodicIdentifier Request Service Id**	**M**	**2A**	**RDBPI**
#2	transmissionMode	M	00-FF	TM
#3	periodicRecordDataIdentifier[] #1	C	00-FF	PRDI1
:	:	:	:	:
#m+2	periodicRecordDataIdentifier[] #m	U	00-FF	PRDIm
C: The first periodicRecordDataIdentifier is mandatory to be present in the request message if the transmissionMode is equal to sendAtSlowRate, sendAtMediumRate, or sendAtFastRate. In case the transmissionMode is equal to stopSending there can either be no periodicRecordDataIdentifier present in order to stop all scheduled periodicRecordDataIdentifier or the client can explicitly specify one or more periodicRecordDataIdentifier(s) to be stopped.				

9.4.2.1 Request message sub-function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

9.4.2.2 Request message data parameter definition

The following data-parameters are defined for this service.

Table 156 — Request message data parameter definition

Definition
transmissionMode This parameter identifies the transmission rate of the requested periodicRecordDataIdentifiers to be used by the server (see annex C.4).
periodicRecordDataIdentifier (#1 to #m) This parameter identifies the server data record(s) that are being requested by the client (see annex C.1 and service description above for detailed parameter definition). It shall be possible to request multiple periodicRecordDataIdentifiers with a single request.

9.4.3 Positive response message definition

It has to be distinguished between the initial positive response message, which indicates that the server accepts the service and subsequent positive response messages, which include periodicRecordDataIdentifier data.

The following table defines the initial positive response message to be transmitted by the server when it accepts the request.

Table 157 —Positive response message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDataByPeriodicIdentifier Response Service Id**	**M**	**6A**	**RDBPIPR**

There are two types of periodic data response messages defined to transmit the periodicRecordDataIdentifier data to the client in order to maximize the useable data portion provided by certain data link layers:

— Response message type #1: Including the service identifier, the echo of the periodicRecordDataIdentifier and the data of the periodicRecordDataIdentifier.

— Response message type #2: Including the periodicRecordDataIdentifier and the data of the periodicRecordDataIdentifier.

A single server shall only support one type of response messages.

The data of a periodicRecordDataIdentifier is transmited periodically. Hereby it has to be distinguished between the following types of data transmissions:

— Single response:

  A single response is sent once: In case a single periodicRecordDataIdentifiers is requested then a single response is sent by the server, containing the actual /current data of the periodicRecordDataIdentifier.

— Multiple responses:

  Multiple responses are a number of responses, each with different data is sent by the server. In case multiple periodicRecordDataIdentifiers are requested then the server responds with multiple responses, where each response of the multiple responses is a single reponse and contains data of a single requested periodicDataIdentifier.

— Periodical responses:

  Periodical responses are a number of responses, each with the same but updated data are sent periodically. In case one single or multiple periodicRecordDataIdentifiers is (are) requested then the server responds with a single response or multiple responses respectively where the single response and each of the multiple responses contains data of a single requested periodicDataIdentifier. The single response / multiple responses are then periodically sent by the server (each with updated data) on a periodic basis specified via the transmissionMode parameter of the request.

For each requested periodicRecordDataIdentifier the server sends a single response message of either type #1 or type #2 as defined below.

Table 158 — Periodic message data definition - type #1

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDataByPeriodicIdentifier Response Service Id**	**M**	**6A**	**RDBPIPR**
#2	periodicRecordDataIdentifier	M	00-FF	PRDI
#3 : #k+2	dataRecord[] = [data#1 : data#k]	M : U	00-FF : 00-FF	DREC_ DATA_1 : DATA_k

Table 159 — Periodic message data definition - type #2

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	periodicRecordDataIdentifier	M	00-FF	PRDI
#2 : #k+2	dataRecord[] = [data#1 : data#k]	M : U	00-FF : 00-FF	DREC_ DATA_1 : DATA_k

9.4.3.1 Positive response message data parameter definition

This service does not support response message data parameters in the positive response message.

The following table defines the periodic message data parameters of the defined periodic data response message types.

Table 160 — Periodic message data parameter definition

Definition
periodicRecordDataIdentifier This parameter references a periodicRecordDataIdentifier from the request message.
dataRecord This parameter is used by the ReadDataByPeriodicIdentifier positive response message to provide the requested data record values to the client. The content of the dataRecord is not defined in this document and is vehicle manufacturer specific.

9.4.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 161 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** This response code shall be sent if the length of the request message is invalid.	M	IMLOIF
22	**conditionsNotCorrect** This response code shall be sent if the operating conditions of the server are not met to perform the required action.	U	CNC
31	**requestOutOfRange** This code shall be sent if 1. one or multiple of the requested periodicRecordDataIdentifier value(s) is (are) not supported by the device (physical addressing only). 2. the client exceeded the maximum number of periodicRecordDataIdentifiers allowed to be requested at a time. 3. the client specified a particular periodicRecordDataIdentifier multiple times within a single request message.	M	ROOR
33	**securityAccessDenied** This code shall be sent if the periodicRecordDataIdentifier is secured and the server is not in an unlocked state.	M	SAD

9.4.5 Message flow example ReadDataByPeriodicIdentifier

This section specifies the conditions to be fulfilled for the example to perform a ReadDataByPeriodicIdentifier service. The client may request a periodicRecordDataIdentifier data at any time independent of the status of the server.

The periodicRecordDataIdentifier examples below are specific to a powertrain device (e.g., engine control module). Refer to ISO/DIS 15031-2 for further details regarding accepted terms/definitions/acronyms for emission related systems.

The example demonstrates requesting of multiple recordDataIdentifiers with a single request (where periodicRecordDataIdentifier E3 hex (= recordDataIdentifier F2E3 hex) contains engine coolant temperature, throttle position, engine speed, vehicle speed sensor, and periodicRecordDataIdentifier 24 hex (= recordDataIdentifier F224 hex) contains battery positive voltage, manifold absolute pressure, mass air flow, vehicle barometric pressure, and calculated load value).

The client requests the transmission at medium rate and after a certain amount of time retrieving the periodic data the client stops the transmission of the periodicRecordDataIdentifier E3 hex only.

For the examples it is assumed that response message type #1 is used to transmit the data of the periodicRecordDataIdentifier.

9.4.5.1 Example: Read multiple periodicRecordDataIdentifiers E3 hex and 24 hex at medium rate

9.4.5.1.1 Step #1: Request periodic transmission of the periodicRecordDataIdentifiers

Table 162 — ReadDataByPeriodicIdentifier request message flow example – step #1

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByPeriodicIdentifier request SID	2A	RDBPI
#2	transmissionMode = sendAtMediumRate	03	TM_SAMR
#3	periodicRecordDataIdentifier #1	E3	PRDI1
#4	periodicRecordDataIdentifier #2	24	PRDI2

Table 163 — ReadDataByPeriodicIdentifier initial positive response message flow example – step #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByPeriodicIdentifier response SID	6A	RDBPIPR

Table 164 — ReadDataByPeriodicIdentifier sub-sequent positive response message #1 flows – step #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByPeriodicIdentifier response SID	6A	RDBPIPR
#2	periodicRecordDataIdentifier #1	E3	PRDI1
#3	dataRecord[data#1] = ECT	A6	DREC_DATA_1
#4	dataRecord[data#2] = TP	66	DREC_DATA_2
#5	dataRecord[data#3] = RPM (high byte)	07	DREC_DATA_3
#6	dataRecord[data#4] = RPM (low byte)	50	DREC_DATA_4
#7	dataRecord[data#5] = VSS	00	DREC_DATA_5

Table 165 — ReadDataByPeriodicIdentifier sub-sequent positive response message #2 flows – step #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByPeriodicIdentifier response SID	6A	RDBPIPR
#2	periodicRecordDataIdentifier #1	24	PRDI2
#3	dataRecord[data#1] = B+	8C	DREC_DATA_1
#4	dataRecord[data#2] = MAP	20	DREC_DATA_2
#5	dataRecord[data#3] = MAF	1A	DREC_DATA_3
#6	dataRecord[data#4] = BARO	63	DREC_DATA_4
#7	dataRecord[data#5] = LOAD	4A	DREC_DATA_5

The server transmits the above shown sub-sequent response messages at the medium rate as applicable to the server.

9.4.5.1.2 Step #2: Stop the transmission of the periodicRecordDataIdentifiers

Table 166 — ReadDataByIdentifier request message flow example – step #2

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByPeriodicIdentifier request SID	2A	RDBPI
#2	transmissionMode = stopSending	04	TM_SS
#3	periodicRecordDataIdentifier #1	E3	PRDI

Table 167 — ReadDataByIdentifier positive response message flow example – step #2

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByPeriodicIdentifier response SID	6A	RDBPIPR

The server stops the transmission of the periodicRecordDataIdentifier E3 hex only. The periodicRecordDataIdentifier 24 hex is still transmitted at the server medium rate.

9.4.5.2 Graphical and tabular example of ReadDataByPeriodicIdentifier service periodic schedule rates

This section contains two examples of scheduled periodic data. Each example contains a graphical and tabular example of the ReadDataByPeriodicIdentifier (2A hex) service. The first example is based on the example given in section 9.4.5.1. The examples contain a graphical depiction of what messages (request/response) are transmitted between the client and the server application, followed by a table which shows a possible implementation of a server periodic scheduler, its variables and how they change each time the background function that checks the periodic scheduler is executed.

In the examples below, the following information is given:

— The fast rate is 25ms and the medium rate is 300 ms.

— The periodic scheduler is checked every 12.5 ms, which means that the periodic scheduler background function is called (polled) with this period.

— The periodic scheduler can hold a maximum of four scheduled items.

— It is possible to send a ReadDataByPeriodicIdentifier response containing a periodicrecordIdentifier any time it's counter has expired.

Since the periodic scheduler poll-rate is 12.5 ms, the fast rate loop counter would be set to 2 (this value is based on the scheduled rate (25 ms) divided by the periodic scheduler poll-rate (12.5 ms) or 25 / 12.5) each time a fast rate periodicRecordIdentifier is sent and the medium rate loop counter would be reset to 24 (scheduled rate divided by the periodic scheduler poll-rate or 300 / 12.5) each time a medium rate periodicRecordIdentifier is sent.

9.4.5.2.1 Example #1: Read multiple periodicRecordDataIdentifiers E3 hex and 24 hex at medium rate

This example is based on the example given in section 9.4.5.1. At t = 0.0 ms, the client begins sending the request to schedule the 2 periodicRecordIdentifier at the medium rate. For the purposes of this example, the server receives the request and executes the periodic scheduler background function the first time t = 25.0 ms.

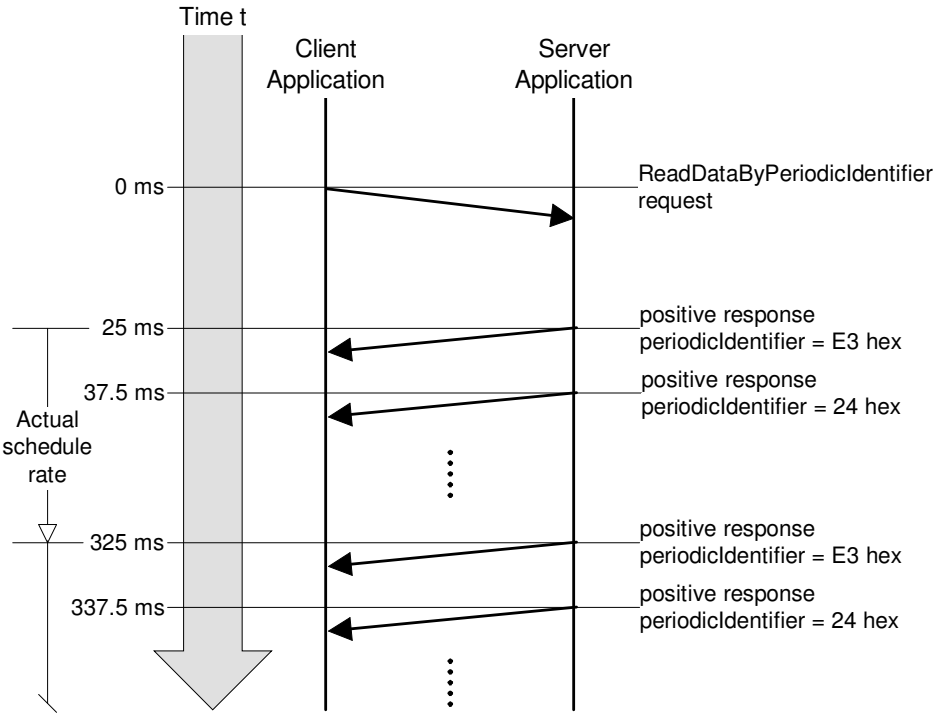


Figure 16 — Example #1: periodicRecordIdentifiers scheduled at medium rate (300 ms)

The following table shows a possible implementation of the periodic scheduler in the server. The table contains the periodic scheduler variables and how they change each time the background function that checks the periodic scheduler is executed.

Table 168 — Example #1: Periodic scheduler table

t (ms)	periodic scheduler transmit Index	periodic identifier sent	periodic scheduler loop #	scheduler[0] Transmit Count	scheduler[1] Transmit Count
25.0	0	1	1	0->24	0
37.5	1	2	2	23	0->24
50.0	0	None	3	22	23
62.5	0	None	4	21	22
75.0	0	None	5	20	21
87.5	0	None	6	19	20
100.0	0	None	7	18	19
112.5	0	None	8	17	18
125.0	0	None	9	16	17
137.5	0	None	10	15	16
150.0	0	None	11	14	15
162.5	0	None	12	13	14
175.0	0	None	13	12	13
187.5	0	None	14	11	12
200.0	0	None	15	10	11
212.5	0	None	16	9	10

Table 168 — Example #1: Periodic scheduler table

t (ms)	periodic scheduler transmit Index	periodic identifier sent	periodic scheduler loop #	scheduler[0] Transmit Count	scheduler[1] Transmit Count
225.0	0	None	17	8	9
237.5	0	None	18	7	8
250.0	0	None	19	6	7
262.5	0	None	20	5	6
275.0	0	None	21	4	5
287.5	0	None	22	3	4
300.0	0	None	23	2	3
312.5	0	None	24	1	2
325.0	0	1	25	0->24	1
337.5	1	2	26	23	0->24
350.0	0	None	27	22	23
362.5	0	None	28	21	22

9.4.5.2.2 Example #2: Read multiple periodicRecordDataIdentifiers at different periodic rates

In this example, three (3) periodicIdentifiers (for simplicity 01 hex, 02 hex, 03 hex) are scheduled at the fast rate and then another request is sent for a single periodicIdentifier (04 hex) to be scheduled at the medium rate. For the purposes of this example, the server receives the first ReadDataByPeriodicIdentifier request and executes the periodic scheduler background function for the first time t = 25.0 ms. The second ReadDataByPeriodicIdentifier request is received and processed just prior to the device executing the periodic scheduler background function at t = 50.0 ms.

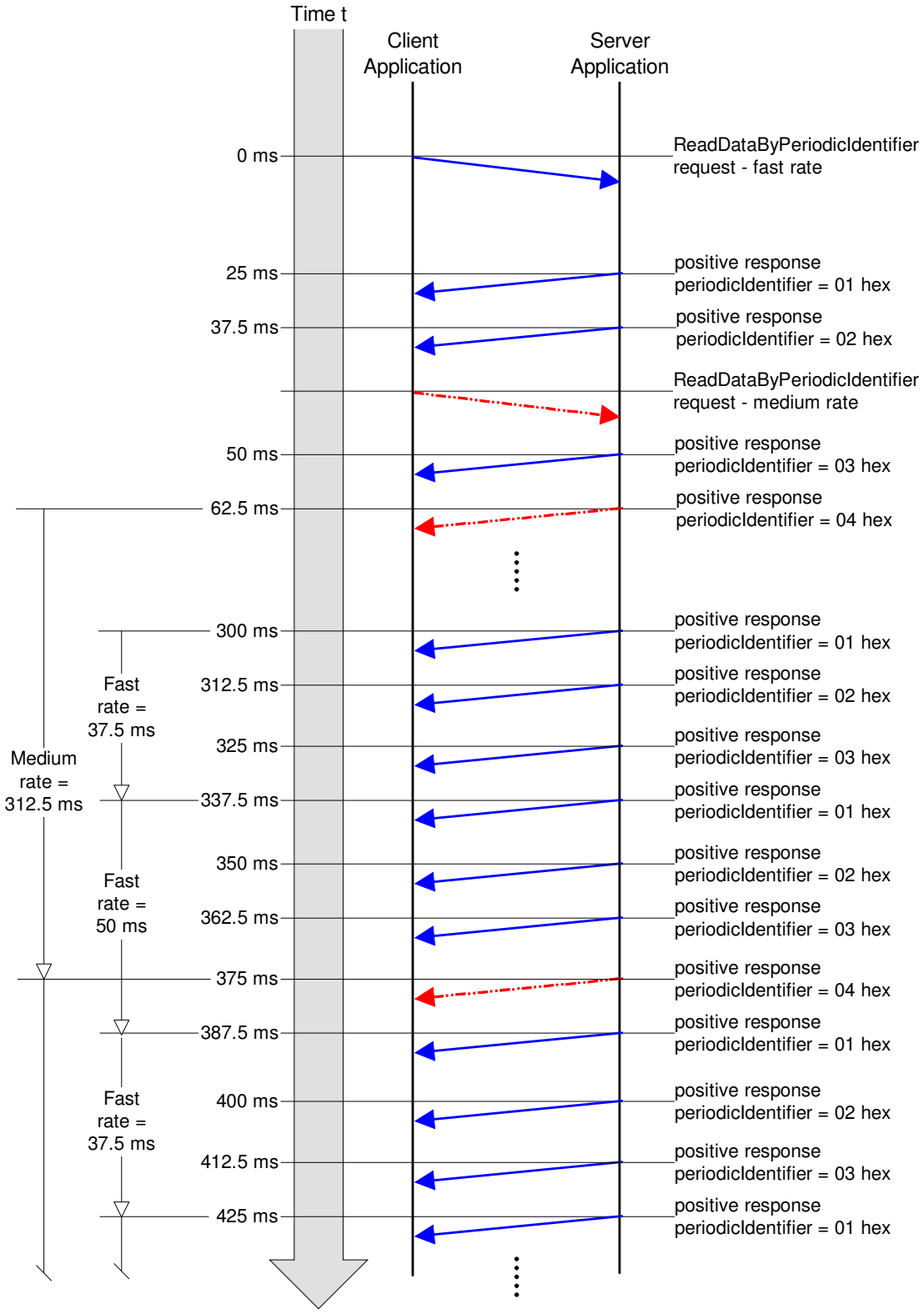


Figure 17 — Example #2: periodicRecordIdentifiers scheduled at fast (25 ms) and medium rate (300 ms)

The following table shows a possible implementation of the periodic scheduler in the server. The table contains the periodic scheduler variables and how they change each time the background function that checks the periodic scheduler is executed.

Table 169 — Example #2: Periodic scheduler table

t (ms)	periodic scheduler transmit Index	periodic identifier sent	periodic scheduler loop #	scheduler[0] Transmit Count	scheduler[1] Transmit Count	scheduler[2] Transmit Count	scheduler[3] Transmit Count
25.0	0	1	1	0->2	0	0	N/A
37.5	1	2	2	1	0->2	0	N/A
50.0	2	3	3	0	1	0->2	0
62.5	3	4	4	0	0	1	0->24
75.0	0	1	5	0->2	0	0	23
87.5	1	2	6	1	0->2	0	22
100.0	2	3	7	0	1	0->2	21
112.5	3	1	8	0->2	0	1	20
125.0	1	2	9	1	0->2	0	19
137.5	2	3	10	0	1	0->2	18
150.0	3	1	11	0->2	0	1	17
162.5	1	2	12	1	0->2	0	16
175.0	2	3	13	0	1	0->2	15
187.5	3	1	14	0->2	0	1	14
200.0	1	2	15	1	0->2	0	13
212.5	2	3	16	0	1	0->2	12
225.0	3	1	17	0->2	0	1	11
237.5	1	2	18	1	0->2	0	10
250.0	2	3	19	0	1	0->2	9
262.5	3	1	20	0->2	0	1	8
275.0	1	2	21	1	0->2	0	7
287.5	2	3	22	0	1	0->2	6
300.0	3	1	23	0->2	0	1	5
312.5	1	2	24	1	0->2	0	4
325.0	2	3	25	0	1	0->2	3
337.5	3	1	26	0->2	0	1	2
350.0	1	2	27	1	0->2	0	1
362.5	2	3	28	0	1	0->2	0
375.0	3	4	29	0	0	1	0->24
387.5	0	1	30	0->2	0	0	23

9.5 DynamicallyDefineDataIdentifier (2C hex) service

The DynamicallyDefineDataIdentifier service allows the client to dynamically define in a server a data identifier that can be read via the ReadDataByIdentifier service at a later time.

9.5.1 Service description

The intention of this service is to provide the client with the ability to group one or more data elements into a data superset that can be requested en masse via the ReadDataByIdentifier or ReadDataByPeriodicIdentifier service. The data elements to be grouped together can either be referenced by:

— a source data identifier, a position and size or,

— a memory address and a memory length, or,

— a combination of the two methods listed above using multiple requests to define the single data element. The dynamically defined recordDataIdentifier will then contain a concatenation of the data parameter definitions.

This service allows greater flexibility in handling ad hoc data needs of the diagnostic application that extend beyond the information that can be read via statically defined data identifiers, and can also be used to reduce bandwidth utilization by avoiding overhead penalty associated with frequent request/response transactions.

The definition of the dynamically defined data identifier can either be done via a single request message or via multiple request messages. This allows for the definition of a single data element referencing source identifier(s) and memory addresses. The server has to concatenate the definitions for the single data element. A redefinition of a dynamically defined data identifier can be achieved by clearing the current definition and start over with the new definition.

Although this service does not prohibit such functionality, it is not recommended for the client to reference one dynamically defined data record from another, because deletion of the referenced record could create data consistency problems within the referencing record.

This service also provides the ability to clear an existing dynamically defined data record. Requests to clear a data record shall be positively responded to if the specified data record identifier exists at the time of the request, and is within the range of valid dynamic data identifiers supported by the server (see annex C.1 for more details).

The server shall maintain the dynamically defined data record until it is cleared. Deletion of the dynamically defined data record upon power down of the server shall be vehicle/manufacturer dependent.

Note that a server can implement data records in two different ways:

1) Composite data records containing multiple elemental data records which are not individually referenced.

2) Unique 2-byte identification “tag” or parameter identifier (PID) value for individual, elemental data records supported within the server (an example elemental data record, or PID, is engine speed or intake air temperature). This implementation of data records is a subset of a composite data record implementation, because it only references a single elemental data record instead of a data record including multiple elemental data records.

Both types of implementing data records are supported by the DynamicallyDefineDataIdentifier service to define a dynamic data identifier.

1) Composite block of data: The position parameter has to reference the starting point in the composite block of data and the size parameter has to reflect the length of data to be placed in the dynamically defined data identifier. It is not possible to only include a portion of an elemental data record of the

composite block of data in the dynamic data record. A request to do this shall be rejected by the server.

2) 2-byte PID: The position parameter has to be set to one (1) and the size parameter has to reflect the length of the PID (length of the elemental data record). It is not possible to only include a portion of the 2-byte PID value in the dynamic data record. A request to do this shall be rejected by the server.

The ordering of the data within the dynamically defined data record shall be of the same order as it was specified in the client request message(s). Also, first position of the data specified in the client's request shall be oriented such that it occurs closest to the beginning of the dynamic data record, in accordance with the ordering requirement mentioned in the preceding sentence.

In addition to the definition of a dynamic data identifier via a logical reference (a record data identifier) this service provides the capability to define a dynamically defined data identifier via an absolute memory address and a memory length information. Note that this mechanism of defining a dynamic data identifier shall only be used during the development phase of a server.

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

9.5.2 Request message definition

Table 170 — Request message definition - sub-function = defineByIdentifier

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**DynamicallyDefineDataIdentifier Request Service Id**	**M**	**2C**	**DDDI**
#2	sub-function = [defineByIdentifier]	M	01	LEV_ DBID
#3 #4	dynamicallyDefinedDataIdentifier[] = [byte 1 (MSB) byte 2]	M M	F2,F3 00-FF	DDDDI_ B1 B2
#5 #6	sourceDataIdentifier[] #1 = [byte 1 (MSB) byte 2]	M M	00-FF 00-FF	SDI_ B1 B2
#7	positionInSourceDataRecord #1	M	00-FF	PISDR#1
#8	memorySize #1	M	00-FF	MS#1
:	:	:	:	:
#n-3 #n-2	sourceDataIdentifier[] #m = [byte 1 (MSB) byte 2]	U U	00-FF 00-FF	SDI_ B1 B2
#n-1	positionInSourceDataRecord #m	U	00-FF	PISDR#m
#n	memorySize #m	U	00-FF	MS#m

Table 171 — Request message definition - sub-function = defineByMemoryAddress

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**DynamicallyDefineDataIdentifier Request Service Id**	**M**	**2C**	**DDDI**
#2	sub-function = [defineByMemoryAddress]	M	02	LEV_ DBMA
#3 #4	dynamicallyDefinedDataIdentifier[] = [byte 1 (MSB) byte 2]	M M	F2,F3 00-FF	DDDDI_ B1 B2
#5	addressAndLengthFormatIdentifier	M_1	00-FF	ALFID
#6 : #(m-1)+3	memoryAddress[] = [byte 1 (MSB) : byte m]	M : C_1	00-FF : 00-FF	MA_ B1 : Bm
#m+4 : #m+4+(k-1)	memorySize[] = [byte 1 (MSB) : byte k]	M : C_2	00-FF : 00-FF	MS_ B1 : Bk
:	:	:	:	:
#n-y-2-(x-1) : #n-(y-1)-1	memoryAddress[] = [byte 1 (MSB) : byte m]	U : U/C_1	00-FF : 00-FF	MA_ B1 : Bm
#n-(y-1) : #n	memorySize[] = [byte 1 (MSB) : byte k]	U : U/C_2	00-FF : 00-FF	MS_ B1 : Bk

M_1: The addressAndLengthFormatIdentifier parameter is only present once at the very beginning of the request message and defines the length of the address and length information for each memory location reference throughout the whole request message.

C_1: The presence of this parameter depends on address length information parameter of the addressAndLengthFormatIdentifier.

C_2: The presence of this parameter depends on the memory size length information of the addressAndLengthFormatIdentifier.

Table 172 — Request message definition - sub-function = clearDynamicallyDefinedDataIdentifier

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**DynamicallyDefineDataIdentifier Request Service Id**	**M**	**2C**	**DDDI**
#2	sub-function = [clearDynamicallyDefinedDataIdentifier]	M	03	LEV_ CDDDID
#3 #4	dynamicallyDefinedDataIdentifier[] = [byte 1 (MSB) byte 2]	M M	F2,F3 00-FF	DDDDI_ B1 B2

9.5.2.1 Request message sub-function parameter $Level (LEV_) definition

The sub-parameters defined as valid for the request message of this service are indicated in the table below (responseRequiredIndicationBit (bit 7) not shown).

Table 173 — Request message sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD
01	**defineByIdentifier** This value shall be used to specify to the server that definition of the dynamic data identifier shall occur via a data identifier reference.	U	DBID
02	**defineByMemoryAddress** This value shall be used to specify to the server that definition of the dynamic data identifier shall occur via an address reference. Note that this sub-function shall only be used during the development phase of the server.	U	DBMA
03	**clearDynamicallyDefinedDataIdentifier** This value shall be used to clear the specified dynamic data identifier. Note that the server shall positively respond to a clear request from the client, even if the specified dynamic data identifier doesn't exist at the time of the request. However, the specified dynamic data identifier is required to be within a valid range (see annex C.1 for allowable ranges).	U	CDDDI
04-7F	**reservedByDocument** This range of values is reserved by this document for future definition.	M	RBD

9.5.2.2 Request message data parameter definition

The following data-parameters are defined for this service.

Table 174 — Request message data parameter definition

Definition
dynamicallyDefinedDataIdentifier This parameter specifies how the dynamic data record, which is being defined by the client, will be referenced in future calls to the service ReadDataByIdentifier or ReadDataByPeriodicDataIdentifier. The dynamicallyDefinedDataIdentifier shall be handled as a recordDataIdentifier in the ReadDataByIdentifier service (see annex C.1 for further details). It shall be handled as a periodicRecordIdentifier in the ReadDataByPeriodicDataIdentifier service (see the ReadDataByPeriodicDataIdentifier service for requirements on the value of this parameter in order to be able to request the dynamically defined data identifier periodically).
sourceDataIdentifier This parameter is only present for sub-function = defineByIdentifier. This parameter logically specifies the source of information to be included into the dynamic data record. For example, this could be a 2-byte PID identifier used to reference engine speed, or a 2-byte data record identifier used to reference a composite block of information containing engine speed, vehicle speed, intake air temperature, etc. (see annex C.1 for further details).
positionInSourceDataRecord This parameter is only present for sub-function = defineByIdentifier. This 1-byte parameter is used to specify the starting position of the excerpt of the source data record to be included in the dynamic data record. A position of one (1) shall reference the first byte of the data record referenced by the sourceDataIdentifier.

Table 174 — Request message data parameter definition

Definition
addressAndLengthFormatIdentifier This parameter is a one byte value with each nibble encoded separately (see annex H.1 for example values): bit 7 - 4: Length (number of bytes) of the memorySize parameter(s) bit 3 - 0: Length (number of bytes) of the memoryAddress parameter(s)
memoryAddress This parameter is only present for sub-function = defineByMemoryAddress. This parameter specifies the memory source address of information to be included into the dynamic data record. The number of bytes used for this address is defined by the low nibble (bit 3 - 0) of the addressFormatIdentifier.
memorySize This parameter is used to specify the total number of bytes from the source data record/memory address that are to be included in the dynamic data record. In case of sub-function = defineByIdentifier then the positionInSourceDataRecord parameter is used in addition to specify the starting position in the source data identifier from where the memorySize applies. The number of bytes used for this size is one (1) byte. In case of sub-function = defineByMemoryAddress then this parameter reflects the number of bytes to be included in the dynamically defined data identifier starting at the specified memoryAddress. The number of bytes used for this size is defined by the high nibble (bit 7 - 4) of the addressFormatIdentifier.

Table 175 —Positive response message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**DynamicallyDefineDataIdentifier Response Service Id**	**M**	**6C**	**DDDIPR**
#2	definitionType	M	00-FF	DM
#3	dynamicallyDefinedDataIdentifier [ ] = [byte 1 (MSB)	M	F2,F3	DDDDI_ B1
#4	byte 2]	M	00-FF	B2

9.5.3 Positive response message data parameter definition

Table 176 — Response message data parameter definition

Definition
definitionType This parameter is an echo of the sub-function parameter from the request message.
dynamicallyDefinedDataIdentifier This parameter is an echo of the data parameter dynamicallyDefinedDataIdentifier from the request message.

9.5.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 177 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** This response code shall be sent if the sub-function parameter is not supported..	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This response code shall be sent if the operating conditions of the server are not met to perform the required action.	M	CNC
31	**requestOutOfRange** This response code shall be sent if: 1. Any data identifier (dynamicallyDefinedDataIdentifier or any sourceDataIdentifier) in the request message is not supported/invalid. 2. The positionInSourceDataRecord was incorrect (less than 1, or greater than maximum allowed by server) 3. Any memory address in the request message is not supported in the server. 4. The specified memorySize was invalid. 5. One or multiple of the specified data portion(s) to be inlcuded in the dynamically defined data identifier only references a portion of an elemental data record. 6. The amount of data to be packed into the dynamic data identifier exceeds the maximum allowed by server. 7. The specified addressAndLengthFormatIdentifier is not valid.	M	ROOR
33	**securityAccessDenied** This code shall be sent if 1. any data identifier (dynamicallyDefinedDataIdentifier or any sourceDataIdentifier) in the request message is secured and the server is not in an unlocked state. 2. any memory address in the request message is secured and the server is not in an unlocked state.	M	SAD

9.5.5 Message flow examples DynamicallyDefineDataIdentifier

This section specifies the conditions to be fulfilled for the example to perform a DynamicallyDefineDataIdentifier service.

The service in this example is not limited by any restriction of the server.

In the first example the server supports 2-byte identifiers (PIDs), which reference a single data information. The example builds a dynamic data identifier using the defineByIdentifier method and then sends a ReadDataByIdentifier request to read the just configured dynamic data identifier.

In the second example the server supports data identifiers, which reference a composite block of data containing multiple data information. The example builds a dynamic identifier also using the defineByIdentifier method, and sends a ReadDataByIdentifier request to read the just defined data identifier.

The third example builds a dynamic data identifier using the defineByMemoryAddress method, and sends a ReadDataByIdentifier request to read the just defined data identifier.

In the fourth example the server supports data identifiers, which reference a composite block of data containing multiple data information. The example builds a dynamic data identifier using the defineByIdentifier method and then uses the ReadDataByPeriodicIdentifier service to requests the dynamically defined data identifier to be sent periodically by the server.

The fifth example demonstrates the deletion of a dynamically defined data identifier.

The following tables shall be used for the examples below. Note that the values being reported may change over time on a real vehicle, but are shown to be constants for the sake of clarity.

Refer to ISO/DIS 15031-2 for further details regarding accepted terms/definitions/acronyms for emissions-related systems.

For all examples the client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

Table 178 — Composite data blocks - DataIdentifier definitions

Data Identifier (Block, hex)	Data Byte	Data Record Contents	Byte Value (Hex)
010A	#1	dataRecord[data#1] = B+	8C
	#2	dataRecord[data#2] = ECT	A6
	#3	dataRecord[data#3] = TP	66
	#4	dataRecord[data#4] = RPM (high byte)	07
	#5	dataRecord[data#5] = RPM (low byte)	50
	#6	dataRecord[data#6] = MAP	20
	#7	dataRecord[data#7] = MAF	1A
	#8	dataRecord[data#8] = VSS	00
	#9	dataRecord[data#9] = BARO	63
	#10	dataRecord[data#10] = LOAD	4A
	#11	dataRecord[data#11] = IAC	82
	#12	dataRecord[data#12] = APP	7E
050B	#1	dataRecord[data#1] = SPARKADV	00
	#2	dataRecord[data#2] = KS	91

Table 179 — Elemental data records - PID definitions

Data Identifier (PID, hex)	Data Byte	Data Record Contents	Byte Value (Hex)
1234	#1	EOT (MSB)	4C
	#2	EOT (LSB)	36
5678	#1	AAT	4D
9ABC	#1	EOL (MSB)	49
	#2	EOL	21
	#3	EOL	00
	#4	EOL (LSB)	17

Table 180 — Memory data records - Memory Address definitions

Memory Address (hex)	Data Byte	Data Record Contents	Byte Value (Hex)
21091968	#1	dataRecord[data#1] = B+	8C
	#2	dataRecord[data#2] = ECT	A6
	#3	dataRecord[data#3] = TP	66
	#4	dataRecord[data#4] = RPM (high byte)	07
	#5	dataRecord[data#5] = RPM (low byte)	50
	#6	dataRecord[data#6] = MAP	20
	#7	dataRecord[data#7] = MAF	1A
	#8	dataRecord[data#8] = VSS	00
	#9	dataRecord[data#9] = BARO	63
	#10	dataRecord[data#10] = LOAD	4A
	#11	dataRecord[data#11] = IAC	82
	#12	dataRecord[data#12] = APP	7E
13101994	#1	dataRecord[data#1] = SPARKADV	00
	#2	dataRecord[data#2] = KS	91

9.5.5.1 Example #1: DynamicallyDefineDataIdentifier, sub-function = defineByIdentifier

This example will build up a dynamic data identifier (DDDI F301 hex) containing engine oil temperature, ambient air temperature, and engine oil level using the 2-byte PIDs as the reference for the required data.

Table 181 — DynamicallyDefineDataIdentifier request DDDDI F301 hex message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier request SID	2C	DDDI
#2	sub-function = defineByIdentifier, responseRequired = yes	01	DBID
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	01	DDDDI_B2
#5	sourceDataIdentifier #1 [byte 1] (MSB) - Engine Oil Temperature	12	SDI_B1
#6	sourceDataIdentifier #1 [byte 2]	34	SDI_B2
#7	positionInSourceDataRecord #1	1	PISDR#1
#8	memorySize #1	2	MS#1
#9	sourceDataIdentifier #2 [byte 1] (MSB) - Ambient Air Temperature	56	SDI_B1
#10	sourceDataIdentifier #2 [byte 2]	78	SDI_B2
#11	positionInSourceDataRecord #2	1	PISDR#2
#12	memorySize #2	1	MS#2
#13	sourceDataIdentifier #3 [byte 1] (MSB) - Engine Oil Level	9A	SDI_B1
#14	sourceDataIdentifier #3 [byte 2]	BC	SDI_B2
#15	positionInSourceDataRecord #3	1	PISDR#3
#16	memorySize #3	4	MS#3

Table 182 — DynamicallyDefineDataIdentifier positive response DDDDI F301 hex message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier response SID	6C	DDDIPR
#2	definitionMode = defineByIdentifier	01	DBID
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	01	DDDDI_B2

Table 183 — ReadDataByIdentifier request DDDDI F301 hex message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier request SID	22	RDBI
#2	recordDataIdentifier [byte 1] (MSB)	F3	RDI_B1
#3	recordDataIdentifier [byte 2]	01	RDI_B2

Table 184 — ReadDataByIdentifier positive response DDDDI F301 hex message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier response SID	62	RDBIPR
#2	recordDataIdentifier [byte 1] (MSB)	F3	RDI_B1
#3	recordDataIdentifier [byte 2]	01	RDI_B2
#4	dataRecord[data#1] = EOT (MSB)	4C	DREC_DATA_1
#5	dataRecord[data#2] = EOT	36	DREC_DATA_2
#6	dataRecord[data#3] = AAT	4D	DREC_DATA_3
#7	dataRecord[data#4] = EOL (MSB)	49	DREC_DATA_4
#8	dataRecord[data#5] = EOL	21	DREC_DATA_5
#9	dataRecord[data#6] = EOL	00	DREC_DATA_6
#10	dataRecord[data#7] = EOL	17	DREC_DATA_7

9.5.5.2 Example #2: DynamicallyDefineDataIdentifier, sub-function = defineByIdentifier

This example will build up a dynamic data identifier (DDDDI F302 hex) containing engine coolant temperature (from data record 010A hex), engine speed (from data record 010A hex), and knock sensor (from data record 050B hex).

Table 185 — DynamicallyDefineDataIdentifier request DDDDI F302 hex message flow example #2

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier request SID	2C	DDDI
#2	sub-function = defineByIdentifier, responseRequired = yes	01	DBID
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	02	DDDDI_B2
#5	sourceDataIdentifier #1 [byte 1] (MSB)	01	SDI_B1
#6	sourceDataIdentifier #1 [byte 2]	0A	SDI_B2
#7	positionInSourceDataRecord #1 - Engine Coolant Temperature	02	PISDR#1
#8	memorySize #1	01	MS#1
#9	sourceDataIdentifier #2 [byte 1] (MSB)	01	SDI_B1
#10	sourceDataIdentifier #2 [byte 2]	0A	SDI_B2
#11	positionInSourceDataRecord #2 - Engine Speed	04	PISDR#2
#12	memorySize #2	02	MS#2
#13	sourceDataIdentifier #3 [byte 1] (MSB)	05	SDI_B1
#14	sourceDataIdentifier #3 [byte 2]	0B	SDI_B2
#15	positionInSourceDataRecord #3 - Knock Sensor	02	PISDR#3
#16	memorySize #3	01	MS#3

Table 186 — DynamicallyDefineDataIdentifier positive response DDDDI F302 hex message flow example #2

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier response SID	6C	DDDIPR
#2	definitionMode = defineByIdentifier	01	DBID
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	02	DDDDI_B2

Table 187 — ReadDataByIdentifier request DDDDI F302 hex message flow example #2

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier request SID	22	RDBI
#2	recordDataIdentifier [byte 1] (MSB)	F3	RDI_B1
#3	recordDataIdentifier [byte 2]	02	RDI_B2

Table 188 — ReadDataByIdentifier positive response DDDDI F302 hex message flow example #2

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier response SID	62	RDBIPR
#2	recordDataIdentifier [byte 1] (MSB)	F3	RDI_B1
#3	recordDataIdentifier [byte 2]	02	RDI_B2
#4	dataRecord[data#1] = ECT	A6	DREC_DATA_1
#5	dataRecord[data#2] = RPM (high byte)	07	DREC_DATA_2
#6	dataRecord[data#3] = RPM (low byte)	50	DREC_DATA_3
#7	dataRecord[data#4] = KS	91	DREC_DATA_4

9.5.5.3 Example #3: DynamicallyDefineDataIdentifier, sub-function = defineByMemoryAddress

This example will build up a dynamic data identifier (DDDDI F302 hex) containing engine coolant temperature (from memory block starting at memory address 21091968 hex), engine speed (from memory block starting at memory address 14121968 hex), and knock sensor (from memory block starting at memory address 13101994 hex).

Table 189 — DynamicallyDefineDataIdentifier request DDDDI F302 hex message flow example #3

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier request SID	2C	DDDI
#2	sub-function = defineByMemoryAddress, responseRequired = yes	02	DBMA
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	02	DDDDI_B2
#5	addressAndLengthFormatIdentifier	14	ALFID
#6	memoryAddress #1 [byte 1] (MSB) - Engine Coolant Temperature	21	MA_B1 #1
#7	memoryAddress #1 [byte 2]	09	MA_B2 #1
#8	memoryAddress #1 [byte 3]	19	MA_B3 #1
#9	memoryAddress #1 [byte 4]	68	MA_B4 #1
#10	memorySize #1	01	MS#1
#11	memoryAddress #2 [byte 1] (MSB) - Engine Speed	14	MA_B1 #2
#12	memoryAddress #2 [byte 2]	12	MA_B2 #2
#13	memoryAddress #2 [byte 3]	19	MA_B3 #2
#14	memoryAddress #2 [byte 4]	68	MA_B4 #2
#15	memorySize #2	02	MS#2
#16	memoryAddress #3 [byte 1] (MSB) - Knock Sensor	13	MA_B1 #3
#17	memoryAddress #3 [byte 2]	10	MA_B2 #3
#18	memoryAddress #3 [byte 3]	19	MA_B3 #3
#19	memoryAddress #3 [byte 4]	94	MA_B4 #3
#20	memorySize #3	01	MS#3

Table 190 — DynamicallyDefineDataIdentifier positive response DDDDI F302 hex message flow example #3

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier response SID	6C	DDDIPR
#2	definitionMode = defineByMemoryAddress	02	DBMA
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	02	DDDDI_B2

Table 191 — ReadDataByIdentifier request DDDDI F302 hex message flow example #3

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier request SID	22	RDBI
#2	recordDataIdentifier [byte 1] (MSB)	F3	RDI_B1
#3	recordDataIdentifier [byte 2]	02	RDI_B2

Table 192 — ReadDataByIdentifier positive response DDDDI F302 hex message flow example #3

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier response SID	62	RDBIPR
#2	recordDataIdentifier [byte 1] (MSB)	F3	RDI_B1
#3	recordDataIdentifier [byte 2]	02	RDI_B2
#4	dataRecord[data#1] = ECT	A6	DREC_DATA_1
#5	dataRecord[data#2] = RPM (high byte)	07	DREC_DATA_2
#6	dataRecord[data#3] = RPM (low byte)	50	DREC_DATA_3
#7	dataRecord[data#4] = KS	91	DREC_DATA_4

9.5.5.4 Example #4: DynamicallyDefineDataIdentifier, sub-function = defineByIdentifier

This example will build up a dynamic data identifier (DDDDI F2E7 hex) containing engine coolant temperature (from data record 010A hex), engine speed (from data record 010A hex), and knock sensor (from data record 050B hex).

The value for the dynamic data identifier is choosen out of the range that can be used to request data periodically. Following the definition of the dynamic data identifier the client requests the data identifier to be sent periodically (fast rate).

Table 193 — DynamicallyDefineDataIdentifier request DDDDI F2E7 hex message flow example #4

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier request SID	2C	DDDI
#2	sub-function = defineByIdentifier, responseRequired = yes	01	DBID
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F2	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	E7	DDDDI_B2
#5	sourceDataIdentifier #1 [byte 1] (MSB)	01	SDI_B1
#6	sourceDataIdentifier #1 [byte 2]	0A	SDI_B2
#7	positionInSourceDataRecord #1 - Engine Coolant Temperature	02	PISDR
#8	memorySize #1	01	MS
#9	sourceDataIdentifier #2 [byte 1] (MSB)	01	SDI_B1
#10	sourceDataIdentifier #2 [byte 2]	0A	SDI_B2
#11	positionInSourceDataRecord #2 - Engine Speed	04	PISDR
#12	memorySize #2	02	MS
#13	sourceDataIdentifier #3 [byte 1] (MSB)	05	SDI_B1
#14	sourceDataIdentifier #3 [byte 2]	0B	SDI_B2
#15	positionInSourceDataRecord #3 - Knock Sensor	02	PISDR
#16	memorySize #3	01	MS

Table 194 — DynamicallyDefineDataIdentifier positive response DDDDI F2E7 hex message flow example #4

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier response SID	6C	DDDIPR
#2	definitionMode = defineByIdentifier	01	DBID
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F2	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	E7	DDDDI_B2

Table 195 — ReadDataByPeriodicIdentifier request DDDDI F2E7 hex message flow example #4

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByPeriodicIdentifier request SID	2A	RDBPI
#2	transmissionMode = sendAtfastRate	04	TM
#3	periodicRecordDataIdentifier	E7	PRDI

Table 196 — ReadDataByPeriodicIdentifier initial positive message flow example #4

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier response SID	6A	RDBPIPR

Table 197 — ReadDataByPeriodicIdentifier positive response #1 DDDDI F2E7 hex message flow example #4

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByPeriodicIdentifier response SID	6A	RDBPIPR
#2	periodicRecordDataIdentifier	E7	PRDI
#3	dataRecord[data#1] = ECT	A6	DREC_DATA_1
#4	dataRecord[data#2] = RPM (high byte)	07	DREC_DATA_2
#5	dataRecord[data#3] = RPM (low byte)	50	DREC_DATA_3
#6	dataRecord[data#4] = KS	91	DREC_DATA_4

:

Table 198 — ReadDataByPeriodicIdentifier positive response #n DDDDI F2E7 hex message flow example #4

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByPeriodicIdentifier response SID	6A	RDBPIPR
#2	periodicRecordDataIdentifier	E7	PRDI
#3	dataRecord[data#1] = ECT	A6	DREC_DATA_1
#4	dataRecord[data#2] = RPM (high byte)	07	DREC_DATA_2
#5	dataRecord[data#3] = RPM (low byte)	55	DREC_DATA_3
#6	dataRecord[data#4] = KS	98	DREC_DATA_4

9.5.5.5 Example #5: DynamicallyDefineDataIdentifier, sub-function = clearDynamicallyDefined-DataIdentifier

This example demonstrates the clearing of a dynamicallyDefinedDataIdentifier, and assumes that DDDDI F303 hex exists at the time of the request.

Table 199 — DynamicallyDefineDataIdentifier request clear DDDDI F303 hex message flow example #5

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier request SID	2C	DDDI
#2	sub-function = clearDynamicallyDefinedDataIdentifier, responseRequired = yes	03	CDDDI
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	03	DDDDI_B2

Table 200 — DynamicallyDefineDataIdentifier positive response clear DDDDI F303 hex message flow example #5

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier response SID	6C	DDDIPR
#2	definitionMode = clearDynamicallyDefinedDataIdentifier	03	CDDDI
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	03	DDDDI_B2

9.5.5.6 Example #6: DynamicallyDefineDataIdentifier, concatenation of definitions (defineByIdentifier / defineByAddress)

This example will build up a dynamic data identifier (DDDI F301 hex) using the two definition types. The following list shows the order of the data in the dynamically defined data identifier (implicit order of request messages to define the dynamic data identifier):

— 1st portion: engine oil temperature and ambient air temperature referecend by 2-byte PIDs (defineByIdentifier),

— 2nd portion: engine coolant temperature and engine speed referenced by memory addresses,

— 3rd portion: engine oil level referecend by 2-byte PID.

9.5.5.6.1 Step #1: DynamicallyDefineDataIdentifier, sub-function = defineByIdentifier (1st portion)

Table 201 — DynamicallyDefineDataIdentifier request DDDI F301 hex message flow example #6 definition of 1st portion (defineByIdentifier)

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier request SID	2C	DDDI
#2	sub-function = defineByIdentifier, responseRequired = yes	01	DBID
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	01	DDDDI_B2
#5	sourceDataIdentifier #1 [byte 1] (MSB) - Engine Oil Temperature	12	SDI_B1
#6	sourceDataIdentifier #1 [byte 2]	34	SDI_B2
#7	positionInSourceDataRecord #1	1	PISDR#1
#8	memorySize #1	2	MS#1
#9	sourceDataIdentifier #2 [byte 1] (MSB) - Ambient Air Temperature	56	SDI_B1
#10	sourceDataIdentifier #2 [byte 2]	78	SDI_B2
#11	positionInSourceDataRecord #2	1	PISDR#2
#12	memorySize #2	1	MS#2

Table 202 — DynamicallyDefineDataIdentifier positive response DDDI F301 hex message flow example #6 definition of first portion (defineByIdentifier)

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier response SID	6C	DDDIPR
#2	definitionMode = defineByIdentifier	01	DBID
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	01	DDDDI_B2

9.5.5.6.2 Step #2: DynamicallyDefineDataIdentifier, sub-function = defineByMemoryAddress (2nd portion)

Table 203 — DynamicallyDefineDataIdentifier request DDDDI F301 hex message flow example #6 definition of 2nd portion (defineByMemoryAddress)

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier request SID	2C	DDDI
#2	sub-function = defineByMemoryAddress, responseRequired = yes	02	DBMA
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	01	DDDDI_B2
#5	addressAndLengthFormatIdentifier	14	ALFID
#6	memoryAddress #1 [byte 1] (MSB) - Engine Coolant Temperature	21	MA_B1 #1
#7	memoryAddress #1 [byte 2]	09	MA_B2 #1
#8	memoryAddress #1 [byte 3]	19	MA_B3 #1
#9	memoryAddress #1 [byte 4]	69	MA_B4 #1
#10	memorySize #1	01	MS#1
#11	memoryAddress #2 [byte 1] (MSB) - Engine Speed	21	MA_B1 #2
#12	memoryAddress #2 [byte 2]	09	MA_B2 #2
#13	memoryAddress #2 [byte 3]	19	MA_B3 #2
#14	memoryAddress #2 [byte 4]	6B	MA_B4 #2
#15	memorySize #2	02	MS#2

Table 204 — DynamicallyDefineDataIdentifier positive response DDDI F301 hex message flow example #6

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier response SID	6C	DDDIPR
#2	definitionMode = defineByMemoryAddress	02	DBMA
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	01	DDDDI_B2

9.5.5.6.3 Step #3: DynamicallyDefineDataIdentifier, sub-function = defineByIdentifier (3rd portion)

Table 205 — DynamicallyDefineDataIdentifier request DDDI F301 hex message flow example #6 definition of 3rd portion (defineByIdentifier)

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier request SID	2C	DDDI
#2	sub-function = defineByIdentifier, responseRequired = yes	01	DBID
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	01	DDDDI_B2
#5	sourceDataIdentifier #1 [byte 1] (MSB) - Engine Oil Level	9A	SDI_B1
#6	sourceDataIdentifier #1 [byte 2]	BC	SDI_B2
#7	positionInSourceDataRecord #1	1	PISDR#3
#8	memorySize #1	4	MS#3

Table 206 — DynamicallyDefineDataIdentifier positive response DDDI F301 hex message flow example #6

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier response SID	6C	DDDIPR
#2	definitionMode = defineByIdentifier	01	DBID
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	01	DDDDI_B2

9.5.5.6.4 Step #4: ReadDataByIdentifier - recordDataIdentifier = DDDDI F301 hex

Table 207 — ReadDataByIdentifier request DDDDI F301 hex message flow example #6

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier request SID	22	RDBI
#2	recordDataIdentifier [byte 1] (MSB)	F3	RDI_B1
#3	recordDataIdentifier [byte 2]	01	RDI_B2

Table 208 — ReadDataByIdentifier positive response DDDDI F301 hex message flow example #6

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier response SID	62	RDBIPR
#2	recordDataIdentifier [byte 1] (MSB)	F3	RDI_B1
#3	recordDataIdentifier [byte 2]	01	RDI_B2
#4	dataRecord[data#1] = EOT (MSB)	4C	DREC_DATA_1
#5	dataRecord[data#2] = EOT	36	DREC_DATA_2
#6	dataRecord[data#3] = AAT	4D	DREC_DATA_3
#7	dataRecord[data#4] = ECT	A6	DREC_DATA_4
#8	dataRecord[data#5] = RPM (high byte)	07	DREC_DATA_5
#9	dataRecord[data#6] = RPM (low byte)	50	DREC_DATA_6
#10	dataRecord[data#7] = EOL (MSB)	49	DREC_DATA_7
#11	dataRecord[data#8] = EOL	21	DREC_DATA_8
#12	dataRecord[data#9] = EOL	00	DREC_DATA_9
#13	dataRecord[data#10] = EOL	17	DREC_DATA_10

9.5.5.6.5 Step #5: DynamicallyDefineDataIdentifier - clear definition of DDDDI F301 hex

Table 209 — DynamicallyDefineDataIdentifier request clear DDDDI F301 hex message flow example #6

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier request SID	2C	DDDI
#2	sub-function = clearDynamicallyDefinedDataIdentifier, responseRequired = yes	03	CDDDI
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	01	DDDDI_B2

Table 210 — DynamicallyDefineDataIdentifier positive response clear DDDDI F301 hex message flow example #6

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	DynamicallyDefineDataIdentifier response SID	6C	DDDIPR
#2	definitionMode = clearDynamicallyDefinedDataIdentifier	03	CDDDI
#3	dynamicallyDefinedDataIdentifier [byte 1] (MSB)	F3	DDDDI_B1
#4	dynamicallyDefinedDataIdentifier [byte 2]	01	DDDDI_B2

9.6 WriteDataByIdentifier (2E hex) service

The WriteDataByIdentifier service allows the client to write information into the server at an internal location specified by the provided data identifier.

9.6.1 Service description

The WriteDataByIdentifier service is used by the client to write a dataRecord to a server. The data is identified by a recordDataIdentifier, and may or may not be secured. It is the vehicle manufacturer's responsibility that the server conditions are met when performing this service. Possible uses for this service are:

— Programming configuration information into server (e.g., VIN number),

— Clearing non-volatile memory,

— Resetting learned values,

— Setting option content.

NOTE The server may restrict or prohibit write access to certain recordDataIdentifier values (as defined by the system supplier / vehicle manufacturer for read-only identifiers, etc.).

9.6.2 Request message definition

Table 211 — Request message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**WriteDataByIdentifier Request Service Id**	**M**	**2E**	**WDBI**
	recordDataIdentifier[] = [			RDI_
#2	byte 1 (MSB)	M	00-FF	B1
#3	byte 2]	M	00-FF	B2
	dataRecord[] = [			DREC_
#4	data#1	M	00-FF	DATA_1
:	:	:	:	:
#m+3	data#m]	U	00-FF	DATA_m

9.6.2.1 Request message sub-function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

9.6.2.2 Request message data parameter definition

The following data-parameters are defined for this service.

Table 212 — Request message data parameter definition

Definition
recordDataIdentifier
This parameter identifies the server data record that the client is requesting to write to (see annex C.1 for detailed parameter definition).
dataRecord
This parameter provides the data record associated with the recordDataIdentifier that the client is requesting to write to.

9.6.3 Positive response message definition

Table 213 —Positive response message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**WriteDataByIdentifier Response Service Id**	**M**	**6E**	**WDBIPR**
	recordDataIdentifier[ ] = [			RDI_
#2	byte 1 (MSB)	M	00-FF	B1
#3	byte 2]	M	00-FF	B2

9.6.3.1 Positive response message data parameter definition

Table 214 — Response message data parameter definition

Definition
recordDataIdentifier
This parameter is an echo of the data parameter recordDataIdentifier from the request message.

9.6.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 215 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This response code shall be sent if the operating conditions of the server are not met to perform the required action.	U	CNC
31	**requestOutOfRange** This response code shall be sent if 1. The recordDataIdentifier in the request message is not supported in the server or the recordDataIdentifier is supported for read only purpose (via ReadDataByIdentifier service) 2. Any data transmitted in the request messager after the recordDataIdentifier is invalid (if applicable to the node)	M	ROOR
33	**securityAccessDenied** This code shall be sent if the recordDataIdentifier, which reference a specific address, is secured and the server is not in an unlocked state..	M	SAD

9.6.5 Message flow example WriteDataByIdentifier

This section specifies the conditions to be fulfilled for the example to perform a WriteDataByIdentifier service.

The service in this example is not limited by any restriction of the server. This example demonstrates VIN programming via recordDataIdentifier F190 hex.

9.6.5.1 Example #1: write recordDataIdentifier F190 hex (VIN)

Table 216 — WriteDataByIdentifier request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	WriteDataByIdentifier request SID	2E	WDBI
#2	recordDataIdentifier [byte 1] (MSB)	F1	RDI_B1
#3	recordDataIdentifier [byte 2]	90	RDI_B2
#4	dataRecord [data_1] = VIN Digit 1 = “W”	57	DREC_DATA1
#5	dataRecord [data_2] = VIN Digit 2 = “0”	30	DREC_DATA2
#6	dataRecord [data_3] = VIN Digit 3 = “L”	4C	DREC_DATA3
#7	dataRecord [data_4] = VIN Digit 4 = “0”	30	DREC_DATA4
#8	dataRecord [data_5] = VIN Digit 5 = “0”	30	DREC_DATA5
#9	dataRecord [data_6] = VIN Digit 6 = “0”	30	DREC_DATA6
#10	dataRecord [data_7] = VIN Digit 7 = “0”	30	DREC_DATA7
#11	dataRecord [data_8] = VIN Digit 8 = “4”	34	DREC_DATA8
#12	dataRecord [data_9] = VIN Digit 9 = “3”	33	DREC_DATA9
#13	dataRecord [data_10] = VIN Digit 10 = “M”	4D	DREC_DATA10
#14	dataRecord [data_11] = VIN Digit 11 = “B”	42	DREC_DATA11
#15	dataRecord [data_12] = VIN Digit 12 = “5”	35	DREC_DATA12
#16	dataRecord [data_13] = VIN Digit 13 = “4”	34	DREC_DATA13
#17	dataRecord [data_14] = VIN Digit 14 = “1”	31	DREC_DATA14
#18	dataRecord [data_15] = VIN Digit 15 = “3”	33	DREC_DATA15
#19	dataRecord [data_16] = VIN Digit 16 = “2”	32	DREC_DATA16
#20	dataRecord [data_17] = VIN Digit 17 = “6”	36	DREC_DATA17

Table 217 — WriteDataByIdentifier positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	WriteDataByIdentifier response SID	6E	WDBIPR
#2	recordDataIdentifier [byte 1] (MSB)	F1	RDI_B1
#3	recordDataIdentifier [byte 2]	90	RDI_B2

9.7 WriteMemoryByAddress (3D hex) service

The WriteMemoryByAddress service allows the client to write information into the server at one or more contiguous memory locations.

9.7.1 Service description

The WriteMemoryByAddress request message writes information specified by the parameter dataRecord[] into the server at memory locations specified by parameters memoryAddress and memorySize. The number of bytes used for the memoryAddress and memorySize parameter is defined by addressAndLengthFormatIdentifier (low and high nibble). It is also possible to use a fixed addressAndLengthFormatIdentifier and unused bytes within the memoryAddress or memorySize parameter are padded with the value 00 hex in the higher range address locations.

The format and definition of the dataRecord shall be vehicle manufacturer specific, and may or may not be secured. It is the vehicle manufacturer's responsibility to assure that the server conditions are met when performing this service. Possible uses for this service are:

— Clear non-volatile memory

— Change calibration values

9.7.2 Request message definition

Table 218 — Request message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**WriteMemoryByAddress Request Service Id**	**M**	**3D**	**WMBA**
#2	addressAndLengthFormatIdentifier	M	00-FF	ALFID
#3 : #m+2	memoryAddress[] = [byte 1 (MSB) : byte m]	M : C_1	00-FF : 00-FF	MA_ B1 : Bm
#n-r-2-(k-1) : #n-r-2	memorySize[] = [byte 1 (MSB) : byte k]	M : C_2	00-FF : 00-FF	MS_ B1 : Bk
#n-(r-1) : #n	dataRecord[] = [data#1 : data#r]	M : U	00-FF : 00-FF	DREC_ DATA_1 : DATA_r
C_1: The presence of this parameter depends on address length information parameter of the addressAndLengthFormatIdentifier				
C_2: The presence of this parameter depends on the memory size length information of the addressAndLengthFormatIdentifier.				

9.7.2.1 Request message sub-function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

9.7.2.2 Request message data parameter definition

The following data-parameters are defined for this service.

Table 219 — Request message data parameter definition

Definition
addressAndLengthFormatIdentifier This parameter is a one byte value with each nibble encoded separately (see annex H.1 for example values): bit 7 - 4: Length (number of bytes) of the memorySize parameter bit 3 - 0: Length (number of bytes) of the memoryAddress parameter
memoryAddress The parameter memoryAddress is the starting address of server memory to which data is to be written. The number of bytes used for this address is defined by the low nibble (bit 3 - 0) of the addressFormatIdentifier. Byte m in the memoryAddress parameter is always the least significant byte of the address being referenced in the server. The most significant byte of the address can be used as a memoryIdentifier. An example of the use of a memoryIdentifier would be a dual processor server with 16 bit addressing and memory address overlap (when a given address is valid for either processor but yields a different physical memory device or internal and external flash is used). In this case, an otherwise unused byte within the memoryAddress parameter can be specified as a memoryIdentifier used to select the desired memory device. Usage of this functionality shall be as defined by vehicle manufacturer / system supplier.
memorySize The parameter memorySize in the ReadMemoryByAddress request message specifies the number of bytes to be read starting at the address specified by memoryAddress in the server's memory. The number of bytes used for this size is defined by the high nibble (bit 7 - 4) of the addressFormatIdentifier.
dataRecord This parameter provides the data that the client is actually attempting to write into the server memory addresses within the interval {$MA, ($MA + $MS - $01)}.

9.7.3 Positive response message definition

Table 220 —Positive response message definition

A_Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**WriteMemoryByAddress Response Service Id**	**M**	**7D**	**WMBAPR**
#2	addressAndLengthFormatIdentifier	M	00-FF	ALFID
#3 : #(m-1)+3	memoryAddress[] = [byte 1 (MSB) : byte m]	M : C_1	00-FF : 00-FF	MA_ B1 : Bm
#n-(k-1) : #n	memorySize[] = [byte 1 (MSB) : byte k]	M : C_2	00-FF : 00-FF	MS_ B1 : Bk
C_1: The presence of this parameter depends on address length information parameter of the addressAndLengthFormatIdentifier				
C_2: The presence of this parameter depends on the memory size length information of the addressAndLengthFormatIdentifier.				

9.7.3.1 Positive response message data parameter definition

Table 221 — Response message data parameter definition

Definition
addressAndLengthFormatIdentifier This parameter is an echo of the addressAndLengthFormatIdentifier from the request message.
memoryAddress This parameter is an echo of the memoryAddress from the request message.
memorySize This parameter is an echo of the memorySize from the request message.

9.7.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 222 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This response code shall be sent if the operating conditions of the server are not met to perform the required action.	U	CNC
31	**requestOutOfRange** 1. Any memory address within the interval [$MA, ($MA + $MS -$1)] is invalid. 2. Any memory address within the interval [$MA, ($MA + $MS -$1)] is restricted. 3. The memorySize parameter value in the request message is greater than the maximum value supported by the server. 4. The specified addressAndLengthFormatIdentifier is not valid.	M	ROOR
33	**securityAccessDenied** This code shall be sent if any memory address within the interval [$MA, ($MA + $MS -$1)] is secure and the server is locked.	M	SAD

9.7.5 Message flow example WriteMemoryByAddress

This section specifies the conditions to be fulfilled for the example to perform a WriteMemoryByAddress service. The service in this example is not limited by any restriction of the server.

The following examples demonstrate writing data bytes into server memory for 2-byte, 3-byte, and 4-byte addressing formats, respectively.

9.7.5.1 Example #1: WriteMemoryByAddress, 2-byte (16-bit) addressing

Table 223 — WriteMemoryByAddress request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	WriteMemoryByAddress request SID	3D	WMBA
#2	addressAndLengthFormatIdentifier	12	ALFID
#3	memoryAddress[byte 1] (MSB)	20	MA_B1
#4	memoryAddress[byte 2] (LSB)	48	MA_B2
#5	memorySize[byte 1]	02	MS_B1
#6	dataRecord [data#1]	00	DREC_DATA_1
#7	dataRecord [data#2]	8C	DREC_DATA_2

Table 224 — WriteMemoryByAddress positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	WriteMemoryByAddress response SID	7D	WMBAPR
#2	addressAndLengthFormatIdentifier	12	ALFID
#3	memoryAddress[byte 1] (MSB)	20	MA_B1
#4	memoryAddress[byte 2] (LSB)	48	MA_B2
#5	memorySize[byte 1]	02	MS_B1

9.7.5.2 Example #2: WriteMemoryByAddress, 3-byte (24-bit) addressing

Table 225 — WriteMemoryByAddress request message flow example #2

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	WriteMemoryByAddress request SID	3D	WMBA
#2	addressAndLengthFormatIdentifier	13	ALFID
#3	memoryAddress[byte 1]	20	MA_B1
#4	memoryAddress[byte 2]	48	MA_B2
#5	memoryAddress[byte 3]	13	MA_B3
#6	memorySize[byte 1]	03	MS_B1
#7	dataRecord [data#1]	00	DREC_DATA_1
#8	dataRecord [data#2]	01	DREC_DATA_2
#9	dataRecord [data#3]	8C	DREC_DATA_3

Table 226 — WriteMemoryByAddress positive response message flow example #2

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	WriteMemoryByAddress response SID	7D	WMBAPR
#2	addressAndLengthFormatIdentifier	13	ALFID
#3	memoryAddress[byte 1]	20	MA_B1
#4	memoryAddress[byte 2]	48	MA_B2
#5	memoryAddress[byte 3]	13	MA_B3
#6	memorySize[byte 1]	03	MS_B1

9.7.5.3 Example #3: WriteMemoryByAddress, 4-byte (32-bit) addressing

Table 227 — WriteMemoryByAddress request message flow example #3

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	WriteMemoryByAddress request SID	3D	WMBA
#2	addressAndLengthFormatIdentifier	14	ALFID
#3	memoryAddress[byte 1] (MSB)	20	MA_B1
#4	memoryAddress[byte 2]	48	MA_B2
#5	memoryAddress[byte 3]	13	MA_B3
#6	memoryAddress[byte 4] (LSB)	09	MA_B4
#7	memorySize[byte 1]	05	MS_B1
#8	dataRecord [data#1]	00	DREC_DATA_1
#9	dataRecord [data#2]	01	DREC_DATA_2
#10	dataRecord [data#3]	8C	DREC_DATA_3
#11	dataRecord [data#4]	09	DREC_DATA_4
#12	dataRecord [data#5]	AF	DREC_DATA_5

Table 228 — WriteMemoryByAddress positive response message flow example #3

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	WriteMemoryByAddress response SID	7D	WMBAPR
#2	addressAndLengthFormatIdentifier	14	ALFID
#3	memoryAddress[byte 1] (MSB)	20	MA_B1
#4	memoryAddress[byte 2]	48	MA_B2
#5	memoryAddress[byte 3]	13	MA_B3
#6	memoryAddress[byte 4] (LSB)	09	MA_B4
#7	memorySize[byte 1]	05	MS_B1

10 Stored Data Transmission functional unit

Table 229 — Stored Data Transmission functional unit

Service	Description
ClearDiagnosticInformation	Allows the client to clear diagnostic information from the server (including DTCs, captured data, etc.)
ReadDTCInformation	Allows the client to request diagnostic information from the server (including DTCs, captured data, etc.)

10.1 ClearDiagnosticInformation (14 hex) Service

The ClearDiagnosticInformation service is used by the client to clear diagnostic information in one or multiple servers' memory.

10.1.1 Service description

The server shall send a positive response when the ClearDiagnosticInformation service is completely processed. The server shall send a positive response even if no DTCs are stored. If a server supports multiple copies of DTC status information in memory (e.g. one copy in RAM and one copy in EEPROM), the server shall clear the copy used by the ReadDTCInformation status reporting service followed by the remaining copy.

The request message of the client contains 1 parameter. The parameter groupOfDTC allows the client to clear a group of DTCs (e.g., Powertrain, Body, Chassis, etc.), or a specific DTC. Refer to annex D.1 for further details.

DTC information reset / cleared via this service includes but is not limited to the following:

— DTC status byte (see ReadDTCInformation service in section 10.2),

— captured DTC snapshot data (DTCSnapShotData, see ReadDTCInformation service in section 10.2),

— captured DTC extended data (DTCExtendedData, see ReadDTCInformation service in section 10.2),

— other DTC related data such as first/most recent DTC, flags, counters, timers, etc. specific to DTCs.

Any DTC information stored in an optionally available DTC mirror memory in the server is not affected by this service (see ReadDTCInformation (19 hex) service in section 10.2 for DTC mirror memory definition).

10.1.2 Request message definition

Table 230 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ClearDiagnosticInformation Request Service Id**	**M**	**14**	**CDI**
	groupOfDTC[] = [			GODTC_
#2	groupOfDTCHighByte	M	00-FF	HB
#3	groupOfDTCMiddleByte	M	00-FF	MB
#4	groupOfDTCLowByte]	M	00-FF	LB

10.1.2.1 Request message sub-function parameter $Level (LEV_) definition

There are no sub-function parameters used by this service.

10.1.2.2 Request message data parameter definition

The following data-parameter is defined for this service.

Table 231 — Request message data parameter definition

Definition
groupOfDTC This parameter contains a 3-byte value indicating the group of DTCs (e.g., Powertrain, Body, Chassis) or the particular DTC to be cleared. The definition of values for each value/range of values is included in annex D.1.

10.1.3 Positive response message definition

Table 232 —Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ClearDiagnosticInformation Positive Response Service Id	M	54	CDIPR

10.1.3.1 Positive response message data parameter definition

There are no data parameters used by this service in the positive response message.

10.1.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service.

Table 233 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This response code shall be used if internal conditions within the server prevent the clearing of DTC related information stored in the server.	C	CNC
31	**requestOutOfRange** This return code shall be sent if the specified groupOfDTC parameter is not supported.	M	ROOR

10.1.5 Message flow example ClearDiagnosticInformation

The client sends a ClearDiagnosticInformation request message to a single server.

Table 234 — ClearDiagnosticInformation request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ClearDiagnosticInformation request SID	14	CDI
#2	groupOfDTC [DTCHighByte] (“Powertrain”)	00	DTCHB
#3	groupOfDTC [DTCMiddleByte]	00	DTCMB
#4	groupOfDTC [DTCLowByte]	00	DTCLB

Table 235 — ClearDiagnosticInformation positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ClearDiagnosticInformation response SID	54	CDIPR

10.2 ReadDTCInformation (19 hex) Service

This service allows a client to read the status of server resident Diagnostic Trouble Code (DTC) information from any server, or group of servers within a vehicle. This service allows the client to do the following:

- Retrieve the status of all DTCs supported by the server.
- Retrieve the number of DTCs matching a client defined DTC status mask (at the point of the request).
- Retrieve DTCSnapshot data associated with a client defined DTC and status mask combination
  - DTC Snapshots are specific data records associated with a DTC, that are stored in the server's memory. The content of the DTC Snapshots is not defined by this standard, but typical usage of DTC Snapshots is to store data upon detection of a system malfunction. The DTC Snapshots will act as a snapshot of data values from the time of the system malfunction occurrence.
- Retrieve DTCExtendedData associated with a client defined DTC and status mask combination out of the DTC memory or the DTC mirror memory.
  - DTC Extended Data consist of extended status information associated with a DTC. DTC Extended Data contains DTC parameter values, which have been identified at the time of the request. A typical use of DTC Extended Data is to store dynamic data associated with the DTC, e.g.
    - DTC occurrence counter
    - current threshold values
    - time of last occurrence (etc.)
    - fault validation counters (e.g. counts number of reported "test failed" and possible other counters if the validation is performed in several steps)
    - uncompleted test counters (e.g. counts numbers of driving cycles since the test was latest completed i.e. since the test reported "test passed" or "test failed"),
    - fault occurrence counters (e.g. counts number of driving cycles in which "testFailed" have been reported),
    - DTC aging counter (e.g. counts number of driving cycles since the fault was latest failed excluding the driving cycles in which the test haven't report "test pass" or the test report "test fail"),
    - specific counters for OBD (e.g. number of remaining driving cycles until the "check engine" lamp is switched off)
- Retrieve the list of DTCs matching a client defined DTC status mask.
- Retrieve the list of DTCs out of the DTC mirror memory matching a client defined DTC status mask.
- Retrieve the most recently failed DTC within the server.
- Retrieve the first DTC failed by the server.
- Retrieve the most recently confirmed DTC within the server.
- Retrieve the first DTC confirmed by the server.

— Retrieve the number of DTCs matching a client defined severity mask (at the point of the request)

— Retrieve the list of DTCs matching a client defined severity mask record

— Retrieve severity information for a client defined DTC

10.2.1 Service description

This service uses a sub-parameter to determine which type of diagnostic information the client is requesting. Further details regarding each sub-function parameter are provided in the following sections.

This service makes use of the following terms:

Enable Criteria:

Server/vehicle manufacturer/system supplier specific criteria used to control when the server actually performs a particular internal diagnostic.

Test Pass Criteria:

Server/vehicle manufacturer/system supplier specific conditions, that define, whether a system being diagnosed is functioning properly within normal, acceptable operating ranges (e.g. no failures exist and the diagnosed system is classified as “OK”).

Test Failure Criteria:

Server/vehicle manufacturer/system supplier specific failure conditions, that define, whether a system being diagnosed has failed the test.

Confirmed Failure Criteria:

Server/vehicle manufacturer/system supplier specific failure conditions that define whether the system being diagnosed is definitively problematic (confirmed), warranting storage of the DTC record in long term memory.

Occurrence Counter:

A counter maintained by certain servers that records the number of instances in which a given DTC test reported a unique occurrence of a test failure.

Aging:

A process whereby certain servers evaluate past results of each internal diagnostic to determine if a confirmed DTC can be cleared from long-term memory, e.g. in the event of a calibrated number of failure free cycles.

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

10.2.1.1 Retrieving the status of all DTCs supported by the server

A client can retrieve the status of all DTCs supported by the server by sending a request for this service with the sub-function set to reportSupportedDTCs. The response to this request contains the DTCStatusAvailabilityMask, which provides an indication of DTC status bits that are supported by the server for masking purposes. Following the DTCStatusAvailabilityMask the response also contains the listOfDTCAndStatusRecord, which contains the DTC number and associated status for every diagnostic trouble code supported by the server.

10.2.1.2 Retrieving the number of DTCs that match a client defined status mask

A client can retrieve a count of the number of DTCs matching a client defined status mask by sending a request for this service with the sub-function set to reportNumberOfDTCByStatusMask. The server shall scan through all supported DTCs, performing a bit-wise logical AND-ing operation between the mask specified by the client with the actual status of each supported DTC. For each AND-ing operation yielding a non-zero result, the server shall increment a counter by 1. If the client specifies a status mask that contains bits that the server does not support, then the server shall process the DTC information using only the bits that it does support. Once all supported DTCs have been checked once, the server shall return the DTCStatusAvailabilityMask and resulting 2-byte count to the client. Note that if no DTCs within the server match the masking criteria specified in the client's request, the count returned by the server to the client shall be 0. The reported number of DTCs matching the DTC status mask is valid for the point in time when the request was made. There is no relationship between the reported number of DTCs and the actual list of DTCs read via the sub-function reportDTCByStatusMask, because the request to read the DTCs is done at a different point in time.

10.2.1.3 Retrieving DTCSnapshot record identification

A client can retrieve DTCSnapshot record identification information for all captured DTCSnapshot records by sending a request for this service with the sub-function set to reportDTCSnapshotIdentification. The server shall return the list of DTCSnapshot record identification information for all stored DTCSnapshot records. Each item the server places in the response message for a single DTCSnapshot record shall contain a DTCRecord (containing the DTC number (high, middle, and low byte)) and the DTCSnapshot record number. In case multiple DTCSnapshot records are stored for a single DTC then the server shall place one item in the response for each occurrence, using a different DTCSnapshot record number for each occurrence (used for the later retrieval of the record data).

NOTE A server may support the storage of multiple DTCSnapshot records for a single DTC to track conditions present at each occurrence of the DTC. Support of this functionality, definition of "occurrence" criteria, and the number of DTCSnapshot records to be supported shall be defined by the system supplier / vehicle manufacturer.

DTCSnapshot record identification information shall be cleared upon a successful ClearDiagnosticInformation request from the client. It is in the responsibility of the vehicle manufacturer to specify the rules for the deletion of stored DTCs and DTCSnapshot data in case of a memory overflow (memory space for stored DTCs and DTCSnapshot data completely occupied in the server).

10.2.1.4 Retrieving DTCSnapshot record data for a client defined DTC mask and/or a client defined DTCSnapshot record number

A client can retrieve captured DTCSnapshot record data for either a client defined DTCMaskRecord in conjunction with a DTCSnapshot record number or a DTCSnapshot record number only by sending a request for this service with the sub-function set to either reportDTCSnapshotRecordByDTCNumber or reportDTCSnapshotRecordByRecordNumber. In case of reportDTCSnapshotRecordByDTCNumber the server shall search through its supported DTCs for an exact match with the DTCMaskRecord specified by the client (containing the DTC number (high, middle, and low byte)). In this case the DTCSnapshotRecordNumber parameter provided in the client's request shall specify a particular occurrence of the specified DTC for which DTCSnapshot record data is being requested. In case of reportDTCSnapshotRecordByRecordNumber the server shall search through its stored DTCSnapshot records for the match of the client provided record number.

NOTE If the DTCSnapshotRecordNumber is unique to the server (each record number exists only once in the server) then both sub-function parameters (reportDTCSnapshotRecordByDTCNumber, reportDTCSnapshotRecordByRecordNumber) to retrieve the DTCSnapshot records can be used. In case the DTCSnapshotRecordNumber is unique to a DTC then only the reportDTCSnapshotRecordByDTCNumber can be used.

If the server supports the ability to store multiple DTCSnapshot records for a single DTC (support of this functionality to be defined by system supplier / vehicle manufacturer), then it is recommended that the server also implements the reportDTCSnapshotRecordIdentification sub-function parameters. It is recommended that the client first requests the identification of DTCSnapshot records stored using the sub-function parameter

reportDTCSnapshotRecordIdentification before requesting a specific DTCSnapshotRecordNumber via the reportDTCSnapshotRecordByDTCNumber or reportDTCSnapshotRecordByRecordNumber.

It is also recommended to support the sub-function parameter reportDTCSnapshotRecordIdentification in order to give the client the opportunity to identify the stored DTCSnapshot records directly instead of parsing through all stored DTCs of the server to determine if a DTCSnapshot record is stored.

It shall be the responsibility of the system supplier / vehicle manufacturer to define whether DTCSnapshot records captured within such servers store data associated with the first or most recent occurrence of a failure.

Along with the DTC number and statusOfDTC, the server shall return a single pre-defined DTCSnapshotRecord in response to the client's request, if a failure has been identified for the client defined DTCMaskRecord and DTCSnapshotRecordNumber parameters (DTCSnapshotRecordNumber unequal FF hex).

NOTE The exact failure criteria shall be defined by the system supplier / vehicle manufacturer.

The DTCSnapshot record may contain multiple data parameters that can be used to reconstruct the vehicle conditions (e.g. B+, RPM, time-stamp) at the time of the failure occurrence.

The vehicle manufacturer shall define format and content of the DTCSnapshotRecord. The data reported in the DTCSnapshotRecord first of all contains a two (2) byte recordDataIdentifier to identify the data that follows. This recordDataIdentifier/data combination can be repeated within the DTCSnapshotRecord.The usage of one or multiple recordDataIdentifiers in the DTCSnapshotRecord allows for the storage of different types of DTCSnapshotRecords for a single DTC for different occurrences of the failure. A parameter which indicates the number of record DataIdentifiers contained within each DTCSnapshotRecord shall be provided with each DTCSnapshotRecord to assist data retreival.

The server shall report one DTCSnapshot record in a single response message, except the client has set the DTCSnapshotRecordNumber to FF hex, because this shall cause the server to response with all DTCSnapshot records stored for the client defined DTCMaskRecord in a single response message.

In case the client requested to report all DTCSnapshot records by DTC number than the DTCAndStatusRecord is only included one time in the ressponse message. In case the client requested to report all DTCSnapshot records by record number then the DTCAndStatusRecord has to be repeated in the response message for each stored DTCSnapshot record.

The server shall negatively respond if the DTCMaskRecord or DTCSnapshotRecordNumber parameters specified by the client are invalid or not supported by the server. This is to be differentiated from the case in which the DTCMaskRecord and/or DTCSnapshotRecordNumber parameters specified by the client are indeed valid and supported by the server, but have no DTCSnapshot data associated with it (e.g., because a failure event never occurred for the specified DTC or record number). In case of reportDTCSnapshotRecordByDTCNumber the server shall send the positive response containing only the DTCAndStatusRecord (echo of the requested DTC number (high, middle, and low byte) plus the statusOfDTC). In case of reportDTCSnapshotRecordByRecordNumber the server shall send the positive response containing only the DTCSnapshotRecordNumber (echo of the requested record number).

DTCSnapshot information shall be cleared upon a successful ClearDiagnosticInformation request from the client. It is in the responsibility of the vehicle manufacturer to specify the rules for the deletion of stored DTCs and DTCSnapshot data in case of a memory overflow (memory space for stored DTCs and DTCsnapshot data completely occupied in the server).

10.2.1.5 Retrieving DTCExtendedData record data for a client defined DTC mask and a client defined DTCExtendedData record number

A client can retrieve DTCExtendedData for a client defined DTCMaskRecord in conjunction with a DTCExtendedDatat record number by sending a request for this service with the sub-function set to reportDTCExtendedDataRecordByDTCNumber. The server shall search through its supported DTCs for an

exact match with the DTCMaskRecord specified by the client (containing the DTC number (high, middle, and low byte)). In this case the DTCExtendedDataRecordNumber parameter provided in the client's request shall specify a particular DTCExtendedData record of the specified DTC for which DTCExtendedData is being requested.

Along with the DTC number and statusOfDTC, the server shall return a single pre-defined DTCExtendedData record in response to the client's request (DTCSnapshotRecordNumber unequal FF hex).

The vehicle manufacturer shall define format and content of the DTCExtendedDataRecord. The structure of the data reported in the DTCExtendedDataRecord is defined by the DTCExtendedDataRecordNumber in a similar way to the definition of data within a record DataIdentifier. Multiple DTCExtendedDataRecordNumbers and associated DTCExtendedDataRecords may be included in the response. The usage of one or multiple DTCExtendedDataRecordNumbers allows for the storage of different types of DTCExtendedDataRecords for a single DTC.

The server shall report one DTCExtendedData record in a single response message, except the client has set the DTCExtendedDataRecordNumber to FF hex, because this shall cause the server to response with all DTCExtendedData records stored for the client defined DTCMaskRecord in a single response message.

The server shall negatively respond if the DTCMaskRecord or DTCExtendedDataRecordNumber parameters specified by the client are invalid or not supported by the server.

Clearance of DTCExtendedData information upon the reception of a ClearDiagnosticInformation service is vehicle manufacturer / system supplier specific. It is in the responsibility of the vehicle manufacturer to specify the rules for the deletion of stored DTCs and DTCSnapshot data in case of a memory overflow (memory space for stored DTCs and DTCsnapshot data completely occupied in the server).

10.2.1.6 Retrieving DTCExtendedData record data for a client defined DTC mask and a client defined DTCExtendedData record number out of the DTC mirror memory

The handling of the sub-function reportMirrorMemoryDTCExtendedDataRecordByDTCNumber is identical to the handling as defined for reportDTCExtendedDataRecordByDTCNumber, except that the data is retrieved out of the DTC mirror memory. The DTC mirror memory is an additional optional error memory in the server that cannot be erased by the ClearDiagnosticInformation (14 hex) service. The DTC mirror memory mirrors the normal DTC memory and can be used for example if the normal error memory is erased.

10.2.1.7 Retrieving the list of DTC's that match a client defined status mask

The client can retrieve a list of DTCs, which satisfy a client defined status mask by sending a request with the sub-function byte set to reportDTCByStatusMask. This sub-function allows the client to request the server to report all DTCs that are "testFailed" OR "confirmed" OR "etc.". The evaluation shall be done as follows: The server shall perform a bit-wise logical AND-ing operation between the mask specified in the client's request and the actual status associated with each DTC supported by the server. In addition to the DTCStatusAvailabilityMask, server shall return all DTC's for which the result of the AND-ing operation is non-zero (i.e., (statusOfDTC & DTCStatusMask) != 0). If the client specifies a status mask that contains bits that the server does not support, then the server shall process the DTC information using only the bits that it does support. If no DTCs within the server match the masking criteria specified in the client's request, no DTC or status information shall be provided following the DTCStatusAvailabilityMask byte in the positive response message.

DTC status information shall be cleared upon a successful ClearDiagnosticInformation request from the client (see DTC status bit definitions in annex D.5 for further descriptions on the DTC status bit handling in case of a ClearDiagnosticInformation service request reception in the server).

10.2.1.8 Retrieving the list of DTC's out of the server DTC mirror memory that match a client defined status mask

The handling of the sub-function reportMirrorMemoryDTCByStatusMask is identical to the handling as defined for reportDTCByStatusMask, except that all status mask checks are performed with the DTCs stored in the

DTC mirror memory of the server. The DTC mirror memory is an additional optional error memory in the server that cannot be erased by the ClearDiagnosticInformation (14 hex) service. The DTC mirror memory mirrors the normal DTC memory and can be used for example if the normal error memory is erased.

10.2.1.9 Retrieving the first / most recent failed DTC

The client can retrieve the first / most recent failed DTC from the server by sending a request with the sub-function byte set to “reportFirstTestFailedDTC” or “reportMostRecentTestFailedDTC”, respectively. Along with the DTCStatusAvailabilityMask, the server shall return the first or most recent failed DTC number and associated status to the client.

No DTC / status information shall be provided following the DTCStatusAvailabilityMask byte in the positive response message if there were no failed DTCs logged since the last time the client requested the server to clear diagnostic information. Also, if only 1 DTC became failed since the last time the client requested the server to clear diagnostic information, the lone failed DTC shall be returned to both reportFirstTestFailedDTC and reportMostRecentTestFailedDTC requests from the client.

Record of the first/most recent failed DTC shall be independent of the aging process of confirmed DTCs.

As mentioned above, first/most recent failed DTC information shall be cleared upon a successful ClearDiagnosticInformation request from the client (see DTC status bit definitions in annex D.5 for further descriptions on the DTC status bit handling in case of a ClearDiagnosticInformation service request reception in the server).

10.2.1.10 Retrieving the first / most recently detected confirmed DTC

The client can retrieve the first / most recent confirmed DTC from the server by sending a request with the sub-function byte set to “reportFirstConfirmedDTC” or “reportMostRecentConfirmedDTC”, respectively. Along with the DTCStatusAvailabilityMask, the server shall return the first or most recent confirmed DTC number and associated status to the client.

No DTC / status information shall be provided following the DTCStatusAvailabilityMask byte in the positive response message if there were no confirmed DTCs logged since the last time the client requested the server to clear diagnostic information. Also, if only 1 DTC became confirmed since the last time the client requested the server to clear diagnostic information, the lone confirmed DTC shall be returned to both reportFirstConfirmedDTC and reportMostRecentConfirmedDTC requests from the client.

The record of the first confirmed DTC shall be preserved in the event that the DTC failed at one point in the past, but then satisfied aging criteria prior to the time of the request from the client (regardless of any other DTCs that become confirmed after the aforementioned DTC became confirmed). Similarly, record of the most recently confirmed DTC shall be preserved in the event that the DTC was confirmed at one point in the past, but then satisfied aging criteria prior to the time of the request from the client (assuming no other DTCs became confirmed after the aforementioned DTC failed).

As mentioned above, first/most recent confirmed DTC information shall be cleared upon a successful ClearDiagnosticInformation request from the client.

10.2.1.11 Retrieving the number of DTC’s that match a client defined severity mask record

A client can retrieve a count of the number of DTCs matching a client defined severity status mask record by sending a request for this service with the sub-function set to reportNumberOfDTCBySeverityMaskRecord. The server shall scan through all supported DTCs, performing a bit-wise logical AND-ing operation between the mask record specified by the client with the actual information of each stored DTC.

```
((statusOfDTC & DTCStatusMask) & (severity & DTCSeverityMask)) != 0
```

For each AND-ing operation yielding a non-zero result, the server shall increment a counter by 1. If the client specifies a status mask within the mask record that contains bits that the server does not support, then the server shall process the DTC information using only the bits that it does support. Once all supported DTCs

have been checked once, the server shall return the DTCStatusAvailabilityMask and resulting 2-byte count to the client.

Note that if no DTCs within the server match the masking criteria specified in the client's request, the count returned by the server to the client shall be 0. The reported number of DTCs matching the DTC status mask is valid for the point in time when the request was made. There is no relationship between the reported number of DTCs and the actual list of DTCs read via the sub-function reportDTCByStatusMask, because the request to read the DTCs is done at a different point in time.

10.2.1.12 Retrieving severity and functional unit information that match a client defined severity mask record

The client can retrieve a list of DTC severity and functional unit information, which satisfy a client defined severity mask record by sending a request with the sub-function byte set to reportDTCBySeverityMaskRecord. This sub-function allows the client to request the server to report all DTCs with a certain severity and status that are ”testFailed” OR ”confirmed” OR ”etc.”. The evaluation shall be done as follows:

The server shall perform a bit-wise logical AND-ing operation between the DTCSeverityMask and the DTCStatusMask specified in the client’s request and the actual DTCSeverity and statusOfDTC associated with each DTC supported by the server.

In addition to the DTCStatusAvailabilityMask, server shall return all DTC’s for which the result of the AND-ing operation is non-zero,

((statusOfDTC & DTCStatusMask) & (severity & DTCSeverityMask)) != 0

If the client specifies a status mask within the mask record that contains bits that the server does not support, then the server shall process the DTC information using only the bits that it does support. If no DTCs within the server match the masking criteria specified in the client’s request, no DTC or status information shall be provided following the DTCStatusAvailabilityMask byte in the positive response message.

10.2.1.13 Retrieving severity and functional unit information for a client defined DTC

A client can retrieve severity and functional unit information for a client defined DTCMaskRecord by sending a request for this service with the sub-function set to reportSeverityInformationOfDTC. The server shall search through its supported DTCs for an exact match with the DTCMaskRecord specified by the client (containing the DTC number (high, middle, and low byte)).

10.2.2 Request message definition

The following tables show the different structures of the ReadDTCInformation request message, based on the used sub-function parameter.

Table 236 — Request message definition - sub-function = reportNumberOfDTC, reportByStatusMask, reportMirrorMemoryDTCByStatusMask

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation request Service Id**	**M**	**19**	**RDI**
#2	sub-function = [reportNumberOfDTCByStatusMask reportDTCByStatusMask reportMirrorMemoryDTCByStatusMask]	M	 01 02 0F	LEV_ RNODTCBSM RDTCBSM RMMDTCBSM
#3	DTCStatusMask	M	00-FF	DTCSM

Table 237 — Request message definition - sub-function = reportDTCSnapshotIdentification, reportDTCSnapshotRecordByDTCNumber

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation request Service Id**	**M**	**19**	**RDI**
#2	sub-function = [reportDTCSnapshotIdentification reportDTCSnapshotRecordByDTCNumber]	M	 03 04	LEV_ RDTCSSI RDTCSSBDTC
 #3 #4 #5	DTCMaskRecord[] = [DTCHighByte DTCMiddleByte DTCLowByte]	 C C C	 00-FF 00-FF 00-FF	DTCMREC_ DTCHB DTCMB DTCLB
#6	DTCSnapshotRecordNumber	C	00-FF	DTCSSRN
C: The DTCMaskRecord record and DTCSnapshotRecordNumber parameters are only present in case the sub-function parameter is equal to reportDTCSnapshotRecordByDTCNumber.				

Table 238 — Request message definition - sub-function = reportDTCSnapshotByRecordNumber

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation request Service Id**	**M**	**19**	**RDI**
#2	sub-function = [reportDTCSnapshotRecordByRecordNumber]	M	 05	LEV_ RDTCSSBRN
#3	DTCSnapshotRecordNumber	M	00-FF	DTCSSRN

Table 239 — Request message definition - sub-function = reportDTCExtendedDataRecordByDTCNumber, reportMirrorMemoryDTCExtendedDataRecordByDTCNumber

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation request Service Id**	**M**	**19**	**RDI**
#2	sub-function = [reportDTCExtendedDataRecordByDTCNumber reportMirrorMemoryDTCExtendedDataRecordByDTCNumber]	M	 06 10	LEV_ RDTCEDRBDN RMDEDRBDN
 #3 #4 #5	DTCMaskRecord[] = [DTCHighByte DTCMiddleByte DTCLowByte]	 M M M	 00-FF 00-FF 00-FF	DTCMREC_ DTCHB DTCMB DTCLB
#6	DTCExtendedDataRecordNumber	M	00-FF	DTCEDRN

Table 240 — Request message definition - sub-function = reportSupportedDTC, reportFirstTestFailedDTC, reportFirstConfirmedDTC, reportMostRecentTestFailedDTC, reportMostRecentConfirmedDTC

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation request Service Id**	**M**	**19**	**RDI**
#2	sub-function = [	M		LEV_
	reportSupportedDTC		0A	RDTCEDRBDN
	reportFirstTestFailedDTC		0B	RFVDTC
	reportFirstConfirmedDTC		0C	RFCDTC
	reportMostRecentTestFailedDTC		0D	RMRVDTC
	reportMostRecentConfirmedDTC]		0E	RMRCDTC

Table 241 — Request message definition - sub-function = reportNumberOfDTCBySeverityMaskRecord, reportDTCSeverityInformation

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation request Service Id**	**M**	**19**	**RDI**
#2	sub-function = [	M		LEV_
	reportNumberOfDTCBySeverityMaskRecord		07	RNODTCBSMR
	reportDTCBySeverityMaskRecord]		08	RDTCBSMR
	DTCSeverityMaskRecord[] = [			DTCSMREC_
#3	DTCSeverityMask	M	00-FF	DTCSM
#4	DTCStatusMask]	M	00-FF	DTCSM

Table 242 — Request message definition - sub-function = reportSeverityInformationOfDTC

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation request Service Id**	**M**	**19**	**RDI**
#2	sub-function = [	M		LEV_
	reportSeverityInformationOfDTC]		09	RSIODTC
	DTCMaskRecord[] = [			DTCMREC_
#3	DTCHighByte	M	00-FF	DTCHB
#4	DTCMiddleByte	M	00-FF	DTCMB
#5	DTCLowByte]	M	00-FF	DTCLB

10.2.2.1 Request message sub-function parameter $Level (LEV_) definition

The sub-function parameters are used by this service to select one of the DTC report types specified in the table below. Explanations and usage of the possible levels are detailed below (responseRequiredIndicationBit (bit 7) not shown).

Table 243 — Request message sub function definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD
01	**reportNumberOfDTCByStatusMask** This parameter specifies that the server shall transmit to the client the number of DTCs matching a client defined status mask.	U	RNODTCBSM

Table 243 — Request message sub function definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
02	**reportDTCByStatusMask** This parameter specifies that the server shall transmit to the client a list of DTCs and corresponding statuses matching a client defined status mask.	M	RDTCBSM
03	**reportDTCSnapshotIdentification** This parameter specifies that the server shall transmit to the client all DTCSnapshot data record identifications (DTC number(s) and DTCSnapshot record number(s)).	U	RDTCSSI
04	**reportDTCSnapshotRecordByDTCNumber** This parameter specifies that the server shall transmit to the client the DTCSnapshot record(s) associated with a client defined DTC number and DTCSnapshot record number (FF hex for all records).	U	RDTCSSBDTC
05	**reportDTCSnapshotRecordByRecordNumber** This parameter specifies that the server shall transmit to the client the DTCSnapshot record(s) associated with a client defined DTCSnapshot record number (FF hex for all records). Note that this sub-function parameter can only be supported if the DTCSnapshotRecordNumber is unique to the server (each record number exists only once in the server) and not unique to a DTC.	U	RDTCSSBRN
06	**reportDTCExtendedDataRecordByDTCNumber** This parameter specifies that the server shall transmit to the client the DTCExtendedData record(s) associated with a client defined DTC number and DTCExtendedData record number (FF hex for all records).	U	RDTCEDRBDN
07	**reportNumberOfDTCBySeverityMaskRecord** This parameter specifies that the server shall transmit to the client the number of DTCs matching a client defined severity mask record.	U	RNODTCBSMR
08	**reportDTCBySeverityMaskRecord** This parameter specifies that the server shall transmit to the client a list of DTCs and corresponding statuses matching a client defined severity mask record.	U	RDTCBSMR
09	**reportSeverityInformationOfDTC** This parameter specifies that the server shall transmit to the client the severity information of a specific DTC specified in the client request message.	U	RSIODTC
0A	**reportSupportedDTC** This parameter specifies that the server shall transmit to the client a list of all DTCs and corresponding statuses supported within the server.	U	RDTCEDRBDN
0B	**reportFirstTestFailedDTC** This parameter specifies that the server shall transmit to the client the first failed DTC to be detected by the server since the last clear of diagnostic information. Note that the information reported via this sub-function parameter shall be independent of whether or not the DTC was confirmed or aged.	U	RFVDTC
0C	**reportFirstConfirmedDTC** This parameter specifies that the server shall transmit to the client the first confirmed DTC to be detected by the server since the last clear of diagnostic information. Note that the information reported via this sub-function parameter shall be independent of the aging process of confirmed DTCs (e.g. if a DTC ages such that its status is allowed to be reset, the first confirmed DTC record shall continue to be preserved by the server, regardless of any other DTCs that become confirmed afterwards).	U	RFCDTC
0D	**reportMostRecentTestFailedDTC** This parameter specifies that the server shall transmit to the client the most recent failed DTC to be detected by the server since the last clear of diagnostic information. Note that the information reported via this sub-function parameter shall be independent of whether or not the DTC was confirmed or aged.	U	RMRVDTC

Table 243 — Request message sub function definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
0E	**reportMostRecentConfirmedDTC** This parameter specifies that the server shall transmit to the client the most recent confirmed DTC to be detected by the server since the last clear of diagnostic information. Note that the information reported via this sub-function parameter shall be independent of the aging process of confirmed DTCs (e.g. if a DTC ages such that its status is allowed to be reset, the first confirmed DTC record shall continue to be preserved by the server assuming no other DTCs become confirmed afterwards).	U	RMRCDTC
0F	**reportMirrorMemoryDTCByStatusMask** This parameter specifies that the server shall transmit to the client a list of DTCs out of the DTC mirror memory and corresponding statuses matching a client defined status mask.	U	RMMDTCBSM
10	**reportMirrorMemoryDTCExtendedDataRecordByDTCNumber** This parameter specifies that the server shall transmit to the client the DTCExtendedData record(s) - out of the DTC mirror memory - associated with a client defined DTC number and DTCExtendedData record number (FF hex for all records) DTCs.	U	RMDEDRBDN
11 - 7F	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

10.2.2.2 Request message data parameter definition

The table below specifies the data parameter definitions for this service.

Table 244 — Request data parameter definition

Definition
DTCStatusMask The DTCStatusMask contains eight (8) DTC status bits. The definitions for each of the eight (8) bits can be found in annex D.5. This byte is used in the request message to allow a client to request DTC information for the DTCs whose status matches the DTCStatusMask. A DTC's status matches the DTCStatusMask if any one of the DTC's actual status bits is set to '1' and the corresponding status bit in the DTCStatusMask is also set to '1' (i.e., if the DTCStatusMask is bit-wise logically ANDed with the DTC's actual status and the result is non-zero, then a match has occurred). If the client specifies a status mask that contains bits that the server does not support, then the server shall process the DTC information using only the bits that it does support.
DTCMaskRecord [DTCHighByte, DTCMiddleByte, DTCLowByte] DTCMaskRecord is a 3-byte value containing DTCHighByte, DTCMiddleByte and DTCLowByte, which together represent a unique identification number for a specific diagnostic trouble code supported by a server. The definition of the 3-byte DTC number allows for several ways of coding DTC information. It can either be done — according to ISO 15031-6, where e.g. the upper two bits of the DTCHighByte are according to ISO 15031-6 or — not according to ISO 15031-6 (typically done for non-OBD servers), where the DTCHighByte and DTCMiddleByte represent a vehicle manufacturer / system supplier specific hexadecimal number and the DTCLowByte represents the DTCFailureTypeByte. The DTCFailureTypeByte is a state-encoded value that is intended to provide a general description of the type of failure that occurred (e.g., open circuit; assignments provided in annex D.2). A DTC within the server matches the DTCMaskRecord if the two values are identical (exact match required).
DTCSnapshotRecordNumber DTCSnapshotRecordNumber is a 1-byte value indicating the number of the specific DTCSnapshot data record requested for a client defined DTCMaskRecord via the reportDTCSnapshotByDTCNumber / reportDTCSnapshotByRecordNumber sub-functions. For servers that do not support multiple DTCSnapshot data records for a single DTC, the client shall set this value to 0. For servers that do support multiple DTCSnapshot data records for a single DTC, the client shall set this to a value ranging from 0 to the maximum number supported by the server (which may range up to FE hex, depending on the server). A value of FF hex requests the server to report all stored DTCSnapshot data records at once.
DTCExtendedDataRecordNumber DTCExtendedDataRecordNumber is a 1-byte value indicating the number of the specific DTCExtendedData record requested for a client defined DTCMaskRecord via the reportDTCExtendedDataRecordByRecordNumber sub-function. For servers that do not support multiple DTCExtendedData records for a single DTC, the client shall set this value to 0. For servers that do support multiple DTCExtendedData records for a single DTC, the client shall set this to a value ranging from 0 to the maximum number supported by the server (which may range up to FE hex, depending on the server). A value of FF hex requests the server to report all stored DTCExtendedData records at once.
DTCSeverityMaskRecord [DTCSeverityMask, DTCStatusMask] DTCSeverityMaskRecord is a 2-byte value containing the DTCSeverityMask and the DTCStatusMask (see annex D.5).
DTCSeverityMask The DTCSeverityMask contains three (3) DTC severity bits. The definitions for each of the three (3) bits can be found in annex D.5. This byte is used in the request message to allow a client to request DTC information for the DTCs whose severity definition matches the DTCSeverityMask. A DTC's severity definition matches the DTCSeverityMask if any one of the DTC's actual severity bits is set to '1' and the corresponding severity bit in the DTCSeverityMask is also set to '1' (i.e., if the DTCSeverityMask is bit-wise logically ANDed with the DTC's actual severity and the result is non-zero, then a match has occurred).

10.2.3 Positive response message definition

Positive response(s) to the service ReadDTCInformation requests depend on the sub-function in the service request.

The tables below define the response message formats of each sub-function parameter.

The following table describes the positive response format for the following sub-functions of this service: reportDTCByStatusMask, reportSupportedDTCs, reportFirstTestFailedDTC, reportFirstConfirmedDTC, reportMostRecentTestFailedDTC, reportMostRecentConfirmedDTC, and reportMirrorMemoryDTCByStatusMask.

Table 245 — Response message definition - sub-function = reportDTCByStatusMask, reportSupportedDTCs, reportFirstTestFailedDTC, reportFirstConfirmedDTC, reportMostRecentTestFailedDTC, reportMostRecentConfirmedDTC, reportMirrorMemoryDTCByStatusMask

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation response Service Id**	**M**	**59**	**RDIPR**
#2	reportType = [	M		LEV_
	reportDTCByStatusMask		02	RDTCBSM
	reportSupportedDTCs		0A	RSDTC
	reportFirstTestFailedDTC		0B	RFVDTC
	reportFirstConfirmedDTC		0C	RFCDTC
	reportMostRecentTestFailedDTC		0D	RMRVDTC
	reportMostRecentConfirmedDTC]		0E	RMRCDTC
#3	DTCStatusAvailabilityMask	M	00-FF	DTCSAM
	DTCAndStatusRecord[] = [			DTCASR_
#4	DTCHighByte#1	C_1	00-FF	DTCHB
#5	DTCMiddleByte#1	C_1	00-FF	DTCMB
#6	DTCLowByte	C_1	00-FF	DTCLB
#7	statusOfDTC#1	C_1	00-FF	SODTC
#8	DTCHighByte#2	C_2	00-FF	DTCHB
#9	DTCLowByte#2	C_2	00-FF	DTCLB
#10	DTCFailureTypeByte	C_2	00-FF	DTCFT
#11	statusOfDTC#2	C_2	00-FF	SODTC
:	:	:	:	:
#n-3	DTCHighByte#m	C_2	00-FF	DTCHB
#n-2	DTCMiddleByte#m	C_2	00-FF	DTCMB
#n-1	DTCLowByte	C_2	00-FF	DTCLB
#n	statusOfDTC#m]	C_2	00-FF	SODTC

C_1: This parameter is only present if reportType = reportSupportedDTCs, reportDTCByStatusMask, reportFirstTestFailedDTC, reportFirstConfirmedDTC, reportMostRecentTestFailedDTC, reportMostRecentConfirmedDTC and DTC information is available to be reported.

C_2: This parameter is only present if if reportType = reportSupportedDTCs, reportDTCByStatusMask and more than one DTC information is available to be reported.

The following table describes the positive response format for the following sub-functions of this service: reportNumberOfDTCByStatusMask, reportNumberOfDTCBySeverityMaskRecord.

Table 246 — Response message definition - sub-function = reportNumberOfDTCByStatusMask, reportNumberOfDTCBySeverityMaskRecord

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation response Service Id**	**M**	**59**	**RDIPR**
#2	reportType = [reportNumberOfDTCByStatusMask reportNumberOfDTCBySeverityMaskRecord]	M	 01 07	LEV_ RNODTCBSM RNODTCBSMR
#3	DTCStatusAvailabilityMask	M	00-FF	DTCSAM
#4	DTCFormatIdentifier = [ISO14229CompliantDTCFormat SAEJ1939CompliantDTCFormat]	M	 00 01	DTCFID_ ISOCDTCF SAECDTCF
 #5 #6	DTCCount[] = [DTCCountHighByte DTCCountLowByte]	 M M	 00-FF 00-FF	DTCC_ DTCCHB DTCCLB

The following table describes the positive response format for the following sub-functions of this service: reportDTCSnapshotIdentification.

Table 247 — Response message definition - sub-function = reportSnapshotIdentification

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation response Service Id**	**M**	**59**	**RDIPR**
#2	reportType = [reportDTCSnapshotIdentification]	M	 03	LEV_ RDTCSSI
 #3 #4 #5	DTCRecord[] #1 = [DTCHighByte#1 DTCMiddleByte#1 DTCLowByte #1]	 C_1 C_1 C_1	 00-FF 00-FF 00-FF	DTCASR_ DTCHB DTCMB DTCLB
#6	DTCSnapshotRecordNumber #1	C_1	00-FF	DTCSSRN
:	:	:	:	:
 #n-3 #n-2 #n-1	DTCRecord[] #m = [DTCHighByte#m DTCMiddleByte#m DTCLowByte #m]	 C_2 C_2 C_2	 00-FF 00-FF 00-FF	DTCASR_ DTCHB DTCMB DTCLB
#n	DTCSnapshotRecordNumber #m	C_2	00-FF	DTCSSRN

C_1: The DTCRecord and DTCSnapshotRecordNumber parameter is only present if at least one DTCSnapshot record is available to be reported.

C_2: The DTCRecord and DTCSnapshotRecordNumber parameter is only present if more than one DTCSnapshot record is available to be reported.

The following table describes the positive response format for the following sub-functions of this service: reportDTCSnapshotRecordByDTCNumber.

Table 248 — Response message definition - sub-function = reportDTCSnapshotRecordByDTCNumber

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation response Service Id**	**M**	**59**	**RDIPR**
#2	reportType = [reportDTCSnapshotRecordByDTCNumber]	M	04	LEV_ RDTCSSBDTC
	DTCAndStatusRecord[] = [			DTCASR_
#3	DTCHighByte	M	00-FF	DTCHB
#4	DTCMiddleByte	M	00-FF	DTCMB
#5	DTCLowByte	M	00-FF	DTCLB
#6	statusOfDTC]	M	00-FF	SODTC
#7	DTCSnapshotRecordNumber #1	C_1	00-FF	DTCSSRN
#8	DTCSnapshotRecordNumberOfIdentifiers #1	C_1	00-FF	DTCSSRNI
	DTCSnapshotRecord[] #1 = [			DTCSSR_
#9	recordDataIdentifier#1 byte #1 (MSB)	C_1	00-FF	RDIB11
#10	recordDataIdentifier#1 byte #2	C_1	00-FF	RDIB12
#11	snapshotData#1 byte #1	C_1	00-FF	SSD11
:	:	C_1	:	:
#11+(p-1)	snapshotData#1 byte #p	C_1	00-FF	SSD1p
:	:	:	:	:
#r-(m-1)-2	recordDataIdentifier#w byte #1 (MSB)	C_2	00-FF	RDIB21
#r-(m-1)-1	recordDataIdentifier#w byte #2	C_2	00-FF	RDIB22
#r-(m-1)	snapshotData#w byte #1	C_2	00-FF	SSD21
:	:	C_2	:	:
#r	snapshotData#w byte #m]	C_2	00-FF	SSD2m
:	:	:	:	:
#t	DTCSnapshotRecordNumber #x	C_3	00-FF	DTCSSRN
#t+1	DTCSnapshotRecordNumberOfIdentifiers #x	C_3	00-FF	DTCSSRNI
	DTCSnapshotRecord[] #x = [			DTCSSR_
#t+2	recordDataIdentifier#1 byte #1 (MSB)	C_3	00-FF	RDIB11
#t+3	recordDataIdentifier#1 byte #2	C_3	00-FF	RDIB12
#t+4	snapshotData#1 byte #1	C_3	00-FF	SSD11
:	:	C_3	:	:
#t+4+(p-1)	snapshotData#1 byte #p	C_3	00-FF	SSD1p
:	:	:	:	:
#n-(u-1)-2	recordDataIdentifier#w byte #1 (MSB)	C_4	00-FF	RDIB21
#n-(u-1)-1	recordDataIdentifier#w byte #2	C_4	00-FF	RDIB22
#n-(u-1)	snapshotData#w byte #1	C_4	00-FF	SSD21
:	:	C_4	:	:
#n	snapshotData#w byte #u]	C_4	00-FF	SSD2u

C_1: The DTCSnapshotRecordNumber and the first recordDataIdentifier/snapshotData combination in the DTCSnapshotRecord parameter is only present if at least one DTCSnapshot record is available to be reported (DTCSnapshotRecordNumber unequal to FF hex in the request or only one record is available to be reported if DTCSnapshotRecordNumber is set to FF hex in the request).

C_2/C_4 There are multiple recordDataIdentifier/snapshotData combinations allowed to be present in a single DTCSnapshotRecord. This can e.g. be the case for the situation where a single recordDataIdentifier only references an integral part of data. When the recordDataIdentifier references a block of data then a single recordDataIdentifier/snapshotData combination can be used.

C_3: The DTCSnapshotRecordNumber and the first recordDataIdentifier/snapshotData combination in the DTCSnapshotRecord parameter is only present if all records are requested to be reported (DTCSnapshotRecordNumber set to FF hex in the request) and more than one record is available to be reported.

The following table describes the positive response format for the following sub-functions of this service: reportDTCSnapshotRecordByRecordNumber.

Table 249 — Response message definition - sub-function = reportDTCSnapshotRecordByRecordNumber

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation response Service Id**	**M**	**59**	**RDIPR**
#2	reportType = [reportDTCSnapshotRecordByRecordNumber]	M	05	LEV_ RDTCSSBRN
#3	DTCSnapshotRecordNumber #1	M	00-FF	DTCEDRN
	DTCAndStatusRecord[] #1 = [			DTCASR_
#4	DTCHighByte	C_1	00-FF	DTCHB
#5	DTCMiddleByte	C_1	00-FF	DTCMB
#6	DTCLowByte	C_1	00-FF	DTCLB
#7	statusOfDTC]	C_1	00-FF	SODTC
#8	DTCSnapshotRecordNumberOfIdentifiers #1	C_1	00-FF	DTCSSRNI
	DTCSnapshotRecord[] #1 = [			DTCSSR_
#9	recordDataIdentifier#1 byte #1 (MSB)	C_1	00-FF	RDIB11
#10#11	recordDataIdentifier#1 byte #2	C_1	00-FF	RDIB12
:	snapshotData#1 byte #1	C_1	00-FF	SSD11
#11+(p-1)	:	C_1	:	:
:	snapshotData#1 byte #p	C_1	00-FF	SSD1p
#r-(m-1)-2	:	:	:	:
#r-(m-1)-1	recordDataIdentifier#w byte #1 (MSB)	C_2	00-FF	RDIB21
#r-(m-1)	recordDataIdentifier#w byte #2	C_2	00-FF	RDIB22
:	snapshotData#w byte #1	C_2	00-FF	SSD21
#r	:	C_2	:	:
	snapshotData#w byte #m]	C_2	00-FF	SSD2m
:	:	:	:	:
#t	DTCSnapshotRecordNumber #x	C_2	00-FF	DTCSSRN
	DTCAndStatusRecord[] #x = [			DTCASR_
#t+1	DTCHighByte	C_2	00-FF	DTCHB
#t+2	DTCMiddleByte	C_2	00-FF	DTCMB
#t+3	DTCLowByte	C_2	00-FF	DTCLB
#t+4	statusOfDTC]	C_2	00-FF	SODTC
#t+5	DTCSnapshotRecordNumberOfIdentifiers #x	C_2	00-FF	DTCSSRNI
	DTCSnapshotRecord[] #x = [			DTCSSR_
#t+6	recordDataIdentifier#1 byte #1 (MSB)	C_3	00-FF	RDIB11
#t+7	recordDataIdentifier#1 byte #2	C_3	00-FF	RDIB12
#t+8	snapshotData#1 byte #1	C_3	00-FF	SSD11
:	:	C_3	:	:
#t+8+(p-1)	snapshotData#1 byte #p	C_3	00-FF	SSD1p
:	:	:	:	:
#n-(u-1)-2	recordDataIdentifier#w byte #1 (MSB)	C_4	00-FF	RDIB21
#n-(u-1)-1	recordDataIdentifier#w byte #2	C_4	00-FF	RDIB22
#n-(u-1)	snapshotData#w byte #1	C_4	00-FF	SSD21
:	:	C_4	:	:
#n	snapshotData#w byte #u]	C_4	00-FF	SSD2u

C_1: The DTCAndStatusRecord and the first recordDataIdentifier/snapshotData combination in the DTCSnapshotRecord parameter is only present if at least one DTCSnapshot record is available to be reported (DTCSnapshotRecordNumber unequal to FF hex in the request or only one record is available to be reported if DTCSnapshotRecordNumber is set to FF hex in the request).

C_2/C_4 There are multiple recordDataIdentifier/snapshotData combinations allowed to be present in a single DTCSnapshotRecord. This can e.g. be the case for the situation where a single recordDataIdentifier only references an integral part of data. When the recordDataIdentifier references a block of data then a single recordDataIdentifier/snapshotData combination can be used.

C_3: The DTCSnapshotRecordNumber, DTCAndStatusRecord, and the first recordDataIdentifier/snapshotData combination in the DTCSnapshotRecord parameter is only present if all records are requested to be reported (DTCSnapshotRecordNumber set to FF hex in the request) and more than one record is available to be reported.

The following table describes the positive response format for the following sub-functions of this service: reportDTCExtendedDataRecordByDTCNumber and reportMirrorMemoryDTCExtendedDataRecordByDTCNumber.

Table 250 — Response message definition - sub-function = reportDTCExtendedDataRecordByDTCNumber and reportMirrorMemoryDTCExtendedDataRecordByDTCNumber

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation response Service Id**	**M**	**59**	**RDIPR**
#2	reportType = [reportDTCExtendedDataRecordByDTCNumber reportMirrorMemoryDTCExtendedDataRecordByDTCNumber]	M	 06 10	LEV_ RDTCEDRBD RMDEDRBDN
 #3 #4 #5 #6	DTCAndStatusRecord[] = [DTCHighByte DTCMiddleByte DTCLowByte statusOfDTC]	 M M M M	 00-FF 00-FF 00-FF 00-FF	DTCASR_ DTCHB DTCMB DTCLB SODTC
#7	DTCExtendedDataRecordNumber #1	C_1	00-FF	DTCEDRN
 #8 : #8+(p-1)	DTCExtendedDataRecord [] #1 = [extendedData #1 byte #1 : extendedData #1 byte #p]	 C_1 C_1 C_1	 00-FF : 00-FF	DTCSSR_ EDD11 : EDD1p
:	:	:	:	:
#t	DTCExtendedDataRecordNumber #x	C_2	00-FF	DTCEDRN
  #t+1 : #t+1+(p-1)	DTCExtendedDataRecord [] #x = [extendedData #x byte #1 : extendedData #x byte #p]	 C_2 C_2 C_2	 00-FF 00-FF 00-FF 00-FF	DTCSSR_ EDDx1 : EDDxp

C_1: The DTCExtendedDataRecordNumber and the extendedData in the DTCExtendedDataRecord parameter are only present if at least one DTCExtendedDatarecord is available to be reported (DTCExtendedDataRecordNumber unequal to FF hex in the request or only one record is available to be reported if DTCExtendedDataRecordNumber is set to FF hex in the request).

C_2: The DTCExtendedDataRecordNumber and the extendedData in the DTCExtendedDataRecord parameter are only present if all records are requested to be reported (DTCExtendedDataRecordNumber set to FF hex in the request) and more than one record is available to be reported.

The following table describes the positive response format for the following sub-functions of this service: reportDTCBySeverityMaskRecord and reportSeverityInformationOfDTC.

Table 251 — Response message definition - sub-function = reportDTCBySeverityMaskRecord, reportSeverityInformationOfDTC

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**ReadDTCInformation response Service Id**	**M**	**59**	**RDIPR**
#2	reportType = [reportDTCBySeverityMaskRecord reportSeverityInformationOfDTC]	M	 08 09	LEV_ RDTCBSMR RSIODTC
#3	DTCStatusAvailabilityMask	M	00-FF	DTCSAM
	DTCAndSeverityRecord[] = [			DTCASR_
#4	DTCSeverity #1	C_1	00-FF	DTCS
#5	DTCFunctionalUnit #1	C_1	00-FF	DTCFU
#6	DTCHighByte #1	C_1	00-FF	DTCHB
#7	DTCMiddleByte #1	C_1	00-FF	DTCMB
#8	DTCLowByte #1	C_1	00-FF	DTCLB
#9	statusOfDTC #1	C_1	00-FF	SODTC
:	:	:	:	:
#n-5	DTCSeverity #m	C_2	00-FF	DTCS
#n-4	DTCFunctionalUnit #m	C_2	00-FF	DTCFU
#n-3	DTCHighByte #m	C_2	00-FF	DTCHB
#n-2	DTCMiddleByte #m	C_2	00-FF	DTCMB
#n-1	DTCLowByte #m	C_2	00-FF	DTCLB
#n	statusOfDTC #m]	C_2	00-FF	SODTC

C_1: This parameter is only present if reportType = reportDTCBySeverityMaskRecord or reportSeverityInformationOfDTC. In case of reportDTCBySeverityMaskRecord this parameter has to be present if at least one DTC matches the client defined DTC severity mask. In case of reportSeverityInformationOfDTC this parameter has to be present if the server supports the DTC specified in the request message.

C_2: This parameter record is only present if reportType = reportDTCBySeverityMaskRecord. It has to be present if more than one DTC matches the client defined DTC severity mask.

10.2.3.1 Positive response message data parameter definition

The table below specifies the response message data parameter definitions for this service.

Table 252 — Response data parameter definition

Definition
reportType This parameter is an echo of the sub-function parameter provided in the request message from the client.
DTCAndSeverityRecord This parameter record contains one or more groupings of DTCSeverity, DTCFunctionalUnit, DTCHighByte, DTCMiddleByte, DTCLowByte, and statusOfDTC if of ISO14229DTCFormat (see below for further details). The DTCSeverity identifies the importance of the failure for the vehicle operation and/or system function and allows to display recommended actions to the driver. The definitions of DTCSeverity's can be found in annex D.3. The DTCFuncitonalUnit is a 1-byte value which identifies the corresponding basic vehicle / system function which reports the DTC. The definitions of DTCFunctionalUnit's can be found in annex D.4. DTCHighByte, DTCMiddleByte and DTCLowByte together represent a unique identification number for a specific diagnostic trouble code supported by a server. The DTCHighByte and DTCMiddleByte represent a circuit or system that is being diagnosed. The DTCLowByte represents the type of fault in the circuit or system (DTCFailureTypeByte, e.g. sensor open circuit, sensor shorted to ground, algorithm based failure, etc). The definition of the DTCFailureTypeByte can be found in annex D.2 of this specification. This parameter record contains one or more groupings of DTCSeverity, DTCFunctionalUnit, SPN (Suspect Parameter Number), FMI (Failure Mode Identifier), and OC (Occurrence Counter) if of SAEJ1939DTCFormat. The SPN, FMI, and OC are defined in SAE J1939.
DTCAndStatusRecord This parameter record contains one or more groupings of either DTCHighByte, DTCMiddleByte, DTCLowByte, and statusOfDTC if of ISO14229DTCFormat (see below for further details) or SPN (Suspect Parameter Number), FMI (Failure Mode Identifier), and OC (Occurrence Counter) if of SAEJ1939DTCFormat. The SPN, FMI, and OC are defined in SAE J1939. DTCHighByte, DTCMiddleByte and DTCLowByte together represent a unique identification number for a specific diagnostic trouble code supported by a server. The coding of the 3-byte DTC number can either be done: — according to ISO 15031-6, where e.g. the upper two bits of the DTCHighByte are according to ISO 15031-6 or it can be done, — not according to ISO 15031-6 (typically done for non-OBD servers), where the DTCHighByte and DTCMiddleByte represent a vehicle manufacturer / system supplier specific hexadecimal number and the DTCLowByte represents the DTCFailureTypeByte. The DTCFailureTypeByte represents the type of fault in the circuit or system (e.g. sensor open circuit, sensor shorted to ground, algorithm based failure, etc). Definition of the DTCFaultTypeByte can be found in annex D.2 of this specification.
DTCRecord This parameter record contains one or more groupings of either DTCHighByte, DTCMiddleByte, and DTCLowByte (DTCFailureTypeByte) if of ISO14229DTCFormat (see DTCAndStatusRecord for further details on the record elements) or SPN (Suspect Parameter Number), FMI (Failure Mode Identifier), and OC (Occurrence Counter) if of SAEJ1939DTCFormat. The SPN, FMI, and OC are defined in SAE J1939.
statusOfDTC The status of a particular DTC. (e.g. DTC failed since power up, passed since power up, etc). The definition of the bits contained in the statusOfDTC byte can be found in annex D.5 of this specification.
DTCStatusAvailabilityMask A byte whose bits are defined the same as statusOfDTC and represents the status bits that are supported by the server. Bits that are not supported by the server shall be set to 0.

Table 252 — Response data parameter definition

Definition
DTCFormatIdentifier A byte which values define the format of a DTC reported by the server. — ISO14229CompliantDTCFormat: This parameter value identifies the DTC format reported by the server as defined in this table by the parameter DTCRecord. — SAEJ1939CompliantDTCFormat: This parameter value identifies the DTC format reported by the server as defined in SAE J1939-73.
DTCCount This 2-byte parameter refers collectively to the DTCCountHighByte and DTCCountLowByte parameters that are sent in response to a reportNumberOfDTC request. DTCCount provides a count of the number of DTC's that match the DTCStatusMask defined in the client's request.
DTCSnapshotRecordNumber Either the echo of the DTCSnapshotRecordNumber parameter specified by the client in the reportDTCSnapshotRecordByDTCNumber / reportDTCSnapshotRecordByRecordNumber request, or the actual DTCSnapshotRecordNumber of a stored DTCSnapshot record.
DTCSnapshotRecordNumberOfIdentifiers This single byte parameter shows the number of recordDataIdentifiers in the immediately following DTCSnapshotRecord.
DTCSnapshotRecord The DTCSnapshotRecord contains a snapshot of data values from the time of the system malfunction occurrence.
DTCExtendedDataRecordNumber Either the echo of the DTCExtendedDataRecordNumber parameter specified by the client in the reportDTCExtendedDataRecordByDTCNumber request, or the actual DTCExtendedDataRecordNumber of a stored DTCExtendedData record.
DTCExtendedDataRecord The DTCExtendedDataRecord is a server -specific block of information that may contain extended status information associated with a DTC. DTCExtendedData contains DTC paramter values, which have been identified at the time of the request.

10.2.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 253 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** This code is returned if the requested sub-function is not supported.	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong.	M	IMLOIF
31	**requestOutOfRange** This code is returned if 1. The client specified a DTCMaskRecord / DTCSeverityMaskRecord that was not recognized by the server. 2. The client specified an invalid DTCSnapshotRecordNumber / DTCExtendedDataRecordNumber (e.g. a record number of 0 hex is not allowed).	M	ROOR

10.2.5 Message flow examples - ReadDTCInformation

The examples below illustrate the operation of each sub-function parameter of the service ReadDTCInformation.

10.2.5.1 Example #1 – ReadDTCInformation - sub-function = reportSupportedDTCs

This example demonstrates the usage of the reportSupportedDTCs sub-function parameter.

For all examples the client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

Assumptions:

a) The server supports a total of 3 DTCs (for the sake of simplicity!), which have the following states at the time of the client request.

b) The following assumptions apply to DTC 123456 hex, statusOfDTC 24 hex (00100100 binary)

- DTC is no longer failed at the time of the request,
- DTC never failed on the current power cycle,
- DTC failed on the previous power cycle (pending DTC),
- DTC was never confirmed,
- DTC tests were completed since the last code clear,
- DTC failed at least once since last code clear,
- DTC test completed this monitoring cycle,
- server does not support warningIndicatorRequested.

c) The following assumptions apply to DTC 234505 hex, statusOfDTC of 00 hex (00000000 binary)

- DTC is not failed at the time of the request,
- DTC never failed on the current power cycle,
- DTC was not failed in this and the previous power cycle (pending DTC),
- DTC is not confirmed at the time of the request,
- DTC tests were completed since the last code clear,
- DTC test never failed since last code clear,
- DTC test completed this monitoring cycle,
- server does not support warningIndicatorRequested.

d) The following assumptions apply to DTC ABCD01 hex, statusOfDTC of 2F hex (00101111 binary)

- DTC failed at the time of the request,

— DTC failed on the current power cycle,

— DTC is confirmed at the time of the request,

— DTC tests were completed since the last code clear,

— DTC test failed at least once since last code clear,

— DTC test completed this monitoring cycle,

— server does not support warningIndicatorRequested.

In the following example, all three of the above DTCs are returned to the client's request because all are supported.

Table 254 — ReadDTCInformation, sub-function = reportSupportedDTCs, request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportSupportedDTCs, responseRequired = yes	0A	RSDTC

Table 255 — ReadDTCInformation, sub-function = readSupportedDTCs, positive response, example #1

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = readSupportedDTCs	0A	RSDTC
#3	DTCStatusAvailabilityMask	2F	DTCSAM
#4	DTCAndStatusRecord #1 [DTCHighByte]	12	DTCHB
#5	DTCAndStatusRecord #1 [DTCMiddleByte]	34	DTCMB
#6	DTCAndStatusRecord #1 [DTCLowByte]	56	DTCLB
#7	DTCAndStatusRecord #1 [statusOfDTC]	24	SODTC
#8	DTCAndStatusRecord #2 [DTCHighByte]	23	DTCHB
#9	DTCAndStatusRecord #2 [DTCMiddleByte]	45	DTCMB
#10	DTCAndStatusRecord #2 [DTCLowByte]	05	DTCLB
#11	DTCAndStatusRecord #2 [statusOfDTC]	00	SODTC
#12	DTCAndStatusRecord #3 [DTCHighByte]	AB	DTCHB
#13	DTCAndStatusRecord #3 [DTCMiddleByte]	CD	DTCMB
#14	DTCAndStatusRecord #3 [DTCLowByte]	01	DTCLB
#15	DTCAndStatusRecord #3 [statusOfDTC]	2F	SODTC

10.2.5.2 Example #2 - ReadDTCInformation, sub-function = reportNumberOfDTCByStatusMask

This example demonstrates the usage of the reportNumberOfDTCByStatusMask sub-function parameter, as well as various masking principles.

Assumptions:

a) The server supports a total of 3 DTCs (for the sake of simplicity!), which have the following states at the time of the client request.

b) The following assumptions apply to DTC 123456 hex, statusOfDTC 24 hex (00100100 binary)

- DTC is no longer failed at the time of the request,
- DTC never failed on the current power cycle,
- DTC failed on the previous power cycle (pending DTC),
- DTC was never confirmed,
- DTC tests were completed since the last code clear,
- DTC failed at least once since last code clear,
- DTC test completed this monitoring cycle,
- server does not support warningIndicatorRequested.

c) The following assumptions apply to DTC 234505 hex, statusOfDTC of 00 hex (00000000 binary)

- DTC is not failed at the time of the request,
- DTC never failed on the current power cycle,
- DTC was not failed in this and the previous power cycle (pending DTC),
- DTC is not confirmed at the time of the request,
- DTC tests were completed since the last code clear,
- DTC test never failed since last code clear,
- DTC test completed this monitoring cycle,
- server does not support warningIndicatorRequested.

d) The following assumptions apply to DTC ABCD01 hex, statusOfDTC of 2F hex (00101111 binary)

- DTC failed at the time of the request,
- DTC failed on the current power cycle,
- DTC is confirmed at the time of the request,
- DTC tests were completed since the last code clear,
- DTC test failed at least once since last code clear,

— DTC test completed this monitoring cycle,

— server does not support warningIndicatorRequested.

In the following example, a count of 1 is returned to the client because only DTC ABCD01 hex status 2F hex matches the client defined status mask of 08 hex (0000 1000 binary).

Table 256 — ReadDTCInformation, sub-function = reportNumberOfDTCByStatusMask, request message flow example #2

Message direction:	client → server		
Message Type:	**Request**		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportNumberOfDTCByStatusMask, responseRequired = yes	01	RNODTCBSM
#3	DTCStatusMask	08	DTCSM

Table 257 — ReadDTCInformation, sub-function = reportNumberOfDTC, positive response, example #2

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportNumberOfDTCByStatusMask	01	RNODTCBSM
#3	DTCStatusAvailabilityMask	2F	DTCSAM
#4	DTCCount [DTCCountHighByte]	00	DTCCHB
#5	DTCCount [DTCCountLowByte]	01	DTCCLB

10.2.5.3 Example #3 - ReadDTCInformation, sub-function = reportDTCSnapshotIdentification

This example demonstrates the usage of the reportDTCSnapshotIdentification sub-function parameter.

Assumptions:

a) The server supports the ability to store 2 DTCSnapshot records for a given DTC.

b) The server shall indicate 2 DTCSnapshot records are currently stored for DTC number 123456 hex. For the purpose of this example, assume that this DTC had occurred three times (such that only the first and most recent DTCSnapshot records are stored because of lack of storage space within the server).

c) The server shall indicate 1 DTCSnapshot record is currently stored for DTC number 789ABC hex.

d) All DTCSnapshot records are stored in ascending order.

Table 258 — ReadDTCInformation, sub-function = reportDTCSnapshotIdentification, request message flow example #3

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportDTCSnapshotIdentification, responseRequired = yes	03	RDTCSSI

Table 259 — ReadDTCInformation, sub-function = reportDTCSnapshotIdentification, positive response, example #3

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportDTCSnapshotIdentification	03	RDTCSSI
#3	DTCAndStatusRecord #1 [DTCHighByte]	12	DTCHB
#4	DTCAndStatusRecord #1 [DTCMiddleByte]	34	DTCMB
#5	DTCAndStatusRecord #1 [DTCLowByte]	56	DTCLB
#6	DTCSnapshotRecordNumber #1	01	DTCEDRC
#7	DTCAndStatusRecord #2 [DTCHighByte]	12	DTCHB
#8	DTCAndStatusRecord #2 [DTCMiddleByte]	34	DTCMB
#9	DTCAndStatusRecord #2 [DTCLowByte]	56	DTCLB
#10	DTCSnapshotRecordNumber #2	02	DTCEDRC
#11	DTCAndStatusRecord #3 [DTCHighByte]	78	DTCHB
#12	DTCAndStatusRecord #3 [DTCMiddleByte]	9A	DTCMB
#13	DTCAndStatusRecord #3 [DTCLowByte]	BC	DTCLB
#14	DTCSnapshotRecordNumber #3	03	DTCEDRC

10.2.5.4 Example #4 - ReadDTCInformation, sub-function = reportDTCSnapshotRecordByDTCNumber

This example demonstrates the usage of the reportDTCSnapshotRecordByDTCNumber sub-function parameter.

Assumptions:

a) The server supports the ability to store 2 DTCSnapshot records for a given DTC.

b) This example assumes a continuation of the previous example.

c) Assume that the server requests the second of the 2 DTCSnapshot records stored by the server for DTC number 123456 hex (see previous example, where a DTCSnapshotRecordCount of 2 is returned to the client).

d) Assume that DTC 123456 hex has a statusOfDTC of 24 hex, and that the following environment data is captured each time a DTC occurs.

e) The DTCSnapshot record data is referenced via the recordDataIdentifier 4711 hex

.

Table 260 — DTCSnapshot record content

Data Byte	DTCSnapshot Record Contents	Byte Value (Hex)
#1	DTCSnapshotRecord [data #1] = ECT (Engine Coolant Temp.)	A6
#2	DTCSnapshotRecord [data #2] = TP (Throttle Position)	66
#3	DTCSnapshotRecord [data #3] = RPM (high byte) (Engine Speed)	07
#4	DTCSnapshotRecord [data #4] = RPM (low byte) (Engine Speed)	50
#5	DTCSnapshotRecord [data #5] = MAP (Manifold Absolute Pressure)	20

Table 261 — ReadDTCInformation, sub-function = reportDTCSnapshotRecordByDTCNumber, request message flow example #4

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportDTCSnapshotRecordByDTCNumber, responseRequired = yes	04	RDTCSSRBDN
#3	DTCMaskRecord [DTCHighByte]	12	DTCHB
#4	DTCMaskRecord [DTCMiddleByte]	34	DTCMB
#5	DTCMaskRecord [DTCLowByte]	56	DTCLB
#6	DTCSnapshotRecordNumber	02	DTCSSRN

Table 262 — ReadDTCInformation, sub-function = reportDTCSnapshotRecordByDTCNumber, positive response, example #4

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportDTCSnapshotRecordByDTCNumber	04	RDTCSSRBDN
#3	DTCAndStatusRecord [DTCHighByte]	12	DTCHB
#4	DTCAndStatusRecord [DTCMiddleByte]	34	DTCMB
#5	DTCAndStatusRecord [DTCLowByte]	56	DTCLB
#6	DTCAndStatusRecord [statusOfDTC]	24	SODTC
#7	DTCSnapshotRecordNumber	02	DTCEDRN
#8	DTCSnapshotRecordNumberOfIdentifiers	01	DTCSSRNI
#9	recordDataIdentifier [byte #1] (MSB)	47	RDIB1
#10	recordDataIdentifier [byte #2]	11	RDIB2
#11	DTCSnapshotRecord [data #1] = ECT	A6	ED_1
#12	DTCSnapshotRecord [data #2] = TP	66	ED_2
#13	DTCSnapshotRecord [data #3] = RPM (high byte)	07	ED_3
#14	DTCSnapshotRecord [data #4] = RPM (low byte)	50	ED_4
#15	DTCSnapshotRecord [data #5] = MAP	20	ED_5

10.2.5.5 Example #5 - ReadDTCInformation, sub-function = reportDTCSnapshotRecordByRecordNumber

This example demonstrates the usage of the reportDTCSnapshotRecordByRecordNumber sub-function parameter.

Assumptions:

a) The server supports the ability to store 2 DTCSnapshot records for a given DTC.

b) This example assumes a continuation of the previous example.

c) Assume that the server requests the second of the 2 DTCSnapshot records stored by the server for DTC number 123456 hex (see previous example, where a DTCSnapshotRecordCount of 2 is returned to the client).

d) Assume that DTC 123456 hex has a statusOfDTC of 24 hex, and that the following environment data is captured each time a DTC occurs.

e) The DTCSnapshot record data is referenced via the recordDataIdentifier 4711 hex.

Table 263 — DTCSnapshot record content

Data Byte	DTCSnapshot Record Contents	Byte Value (Hex)
#1	DTCSnapshotRecord [data #1] = ECT (Engine Coolant Temp.)	A6
#2	DTCSnapshotRecord [data #2] = TP (Throttle Position)	66
#3	DTCSnapshotRecord [data #3] = RPM (high byte) (Engine Speed)	07
#4	DTCSnapshotRecord [data #4] = RPM (low byte) (Engine Speed)	50
#5	DTCSnapshotRecord [data #5] = MAP (Manifold Absolute Pressure)	20

Table 264 — ReadDTCInformation, sub-function = reportDTCSnapshotRecordByRecordNumber, request message flow example #5

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportDTCSnapshotRecordByRecordNumber, responseRequired = yes	05	RDTCSSRBRN
#3	DTCSnapshotRecordNumber	02	DTCSSRN

Table 265 — ReadDTCInformation, sub-function = reportDTCSnapshotRecordByRecordNumber, positive response, example #5

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportDTCSnapshotRecordByRecordNumber	04	RDTCSSRBRN
#3	DTCSnapshotDataRecordNumber	02	DTCSSRN
#4	DTCAndStatusRecord [DTCHighByte]	12	DTCHB
#5	DTCAndStatusRecord [DTCMiddleByte]	34	DTCMB
#6	DTCAndStatusRecord [DTCLowByte]	56	DTCLB
#7	DTCAndStatusRecord [statusOfDTC]	24	SODTC
#8	DTCSnapshotRecordNumberOfIdentifiers	01	DTCSSRNI
#9	recordDataIdentifier [byte #1 (MSB)]	47	RDIB1
#10	recordDataIdentifier [byte #2]	11	RDIB2
#11	DTCSnapshotRecord [data #1] = ECT	A6	ED_1
#12	DTCSnapshotRecord [data #2] = TP	66	ED_2
#13	DTCSnapshotRecord [data #3] = RPM (high byte)	07	ED_3
#14	DTCSnapshotRecord [data #4] = RPM (low byte)	50	ED_4
#15	DTCSnapshotRecord [data #5] = MAP	20	ED_5

10.2.5.6 Example #6 - ReadDTCInformation, sub-function = reportDTCByStatusMask, matching DTCs returned

This example demonstrates usage of the reportDTCByStatusMask sub-function parameter, as well as various masking principles in conjunction with unsupported masking bits. This example also applies to the sub-function parameter reportMirrorMemoryDTCByStatusMask, except that the status mask checks are performed with the DTCs stored in the DTC mirror memory.

Assumptions:

a) The server supports all status bits for masking purposes, except for bit 7 “warningIndicatorRequested”.

b) The server supports a total of 3 DTCs (for the sake of simplicity!), which have the following states at the time of the client request.

c) The following assumptions apply to DTC 123456 hex, statusOfDTC 24 hex (00100100 binary):

- DTC is no longer failed at the time of the request,
- DTC never failed on the current power cycle,
- DTC failed on the previous power cycle (pending DTC),
- DTC was never confirmed,
- DTC tests were completed since the last code clear,
- DTC failed at least once since last code clear,
- DTC test completed this monitoring cycle,
- server does not support warningIndicatorRequested.

d) The following assumptions apply to DTC 234505 hex, statusOfDTC of 00 hex (00000000 binary):

- DTC is not failed at the time of the request,
- DTC never failed on the current power cycle,
- DTC was not failed in this and the previous power cycle (pending DTC),
- DTC is not confirmed at the time of the request,
- DTC tests were completed since the last code clear,
- DTC test never failed since last code clear,
- DTC test completed this monitoring cycle,
- server does not support warningIndicatorRequested.

e) The following assumptions apply to DTC ABCD01 hex, statusOfDTC of 2F hex (00101111 binary):

- DTC is failed at the time of the request,
- DTC failed on the current power cycle,
- DTC is confirmed at the time of the request,

— DTC tests were completed since the last code clear,

— DTC test failed at least once since last code clear,

— DTC test completed this monitoring cycle,

— server does not support warningIndicatorRequested.

In the following example, DTCs 123456 hex and ABCD01 hex are returned to the client's request. DTC 234505 hex is not returned because its status of 00 hex does not match the DTCStatusMask of 84 hex (as specified in the client request messsage in the following example). Note that the server shall bypass masking on those status bits it doesn't support.

Table 266 — ReadDTCInformation, sub-function = reportDTCByStatusMask, request message flow example #6

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportDTCByStatusMask, responseRequired = yes	02	RDTCBSM
#3	DTCStatusMask	84	DTCSM

Table 267 — ReadDTCInformation, sub-function = reportDTCByStatusMask, positive response, example #6

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportDTCByStatusMask	02	RDTCBSM
#3	DTCStatusAvailabilityMask	7F	DTCSAM
#4	DTCAndStatusRecord #1 [DTCHighByte]	12	DTCHB
#5	DTCAndStatusRecord #1 [DTCMiddleByte]	34	DTCMB
#6	DTCAndStatusRecord #1 [DTCLowByte]	56	DTCLB
#7	DTCAndStatusRecord #1 [statusOfDTC]	24	SODTC
#4	DTCAndStatusRecord #2 [DTCHighByte]	AB	DTCHB
#5	DTCAndStatusRecord #2 [DTCMiddleByte]	CD	DTCMB
#6	DTCAndStatusRecord #2 [DTCLowByte]	01	DTCLB
#7	DTCAndStatusRecord #2 [statusOfDTC]	2F	SODTC

10.2.5.7 Example #7 - ReadDTCInformation, sub-function = reportDTCByStatusMask, matching DTCs returned

This example demonstrates usage of the reportDTCByStatusMask sub-function parameter, in the situation where no DTCs match the client defined DTCStatusMask.

Assumptions:

a) The server supports all status bits for masking purposes, except for bit 7 “warningIndicatorRequested”.

b) The server supports a total of 2 DTCs (for the sake of simplicity!), which have the following states at the time of the client request.

c) The following assumptions apply to DTC 123456 hex, statusOfDTC 24 hex (00100100 binary):

- DTC is no longer failed at the time of the request,
- DTC never failed on the current power cycle,
- DTC failed on the previous power cycle (pending DTC),
- DTC was never confirmed,
- DTC tests were completed since the last code clear,
- DTC failed at least once since last code clear,
- DTC test completed this monitoring cycle,
- server does not support warningIndicatorRequested.

d) The following assumptions apply to DTC 234505 hex, statusOfDTC of 00 hex (00000000 binary):

- DTC is not failed at the time of the request,
- DTC never failed on the current power cycle,
- DTC was not failed in this and the previous power cycle (pending DTC),
- DTC is not confirmed at the time of the request,
- DTC tests were completed since the last code clear,
- DTC test never failed since last code clear,
- DTC test completed this monitoring cycle,
- server does not support warningIndicatorRequested.

e) The client requests the server to reportByStatusMask all DTCs having bit 0 (TestFailed) set to logical ‘1’.

Table 268 — ReadDTCInformation, sub-function = reportDTCByStatusMask, request message flow example #7

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportDTCByStatusMask, responseRequired = yes	02	RDTCBSM
#3	DTCStatusMask	01	DTCSM

Table 269 — ReadDTCInformation, sub-function = reportDTCByStatusMask, positive response, example #7

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportDTCByStatusMask	02	RDTCBSM
#3	DTCStatusAvailabilityMask	7F	DTCSAM

10.2.5.8 Example #8 - ReadDTCInformation, sub-function = reportFirstTestFailedDTC, information available

This example demonstrates usage of the reportFirstTestFailedDTC sub-function parameter, where it is assumed that at least 1 failed DTC occurred since the last ClearDiagnosticInformation request from the server.

NOTE 1 If exactly 1 DTC failed within the server since the last ClearDiagnosticInformation request from the server, then the server would return the same information in response to a reportMostRecentTestFailedDTC request from the client.

NOTE 2 In this example, the status of the DTC returned in response to the reportFirstTestFailedDTC is no longer current at the time of the request (the same phenomenon is possible when requesting the server to report the most recent failed / confirmed DTC).

NOTE 3 The general format of request/response messages in the following example is applicable to sub-function parameters reportFirstConfirmedDTC, reportMostRecentTestFailedDTC, and reportMostRecentConfirmedDTC (for the appropriate DTC status and under similar assumptions).

Assumptions:

a) At least 1 DTC failed since the last ClearDiagnosticInformation request from the server.

b) The server supports all status bits for masking purposes.

c) DTC number 123456 hex = first failed DTC to be detected since the last code clear.

d) The following assumptions apply to DTC 123456 hex, statusOfDTC 34 hex (00110100 binary):

— DTC is no longer failed at the time of the request,

— DTC never failed on the current power cycle,

— DTC was failed on the previous power cycle (pending DTC),

— DTC was never confirmed,

— DTC tests were completed since the last code clear,

— DTC failed at least once since last code clear,

— DTC test completed this monitoring cycle,

— server does not support warningIndicatorRequested.

Table 270 — ReadDTCInformation, sub-function = reportFirstTestFailedDTC, request message flow example #8

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportFirstTestFailedDTC, responseRequired = yes	0B	RFCDTC

Table 271 — ReadDTCInformation, sub-function = reportFirstTestFailedDTC, positive response, example #8

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportFirstTestFailedDTC	0B	RFCDTC
#3	DTCStatusAvailabilityMask	FF	DTCSAM
#4	DTCAndStatusRecord [DTCHighByte]	12	DTCHB
#5	DTCAndStatusRecord [DTCMiddleByte]	34	DTCMB
#6	DTCAndStatusRecord [DTCLowByte]	56	DTCLB
#7	DTCAndStatusRecord [statusOfDTC]	34	SODTC

10.2.5.9 Example #9 - ReadDTCInformation, sub-function = reportFirstTestFailedDTC, no information available

This example demonstrates usage of the reportFirstTestFailedDTC sub-function parameter, where it is assumed that no failed DTCs have occurred since the last ClearDiagnosticInformation request from the server.

NOTE The general format of request/response messages in the following example is applicable to sub-function parameters reportFirstConfirmedDTC, reportMostRecentTestFailedDTC, and reportMostRecentConfirmedDTC (for the appropriate DTC status and under similar assumptions).

Assumptions:

a) No failed DTCs have occurred since the last ClearDiagnosticInformation request from the server.

b) The server supports all status bits for masking purposes.

Table 272 — ReadDTCInformation, sub-function = reportFirstTestFailedDTC, request message flow example #9

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportFirstTestFailedDTC, responseRequired = yes	0B	RFCDTC

Table 273 — ReadDTCInformation, sub-function = reportFirstTestFailedDTC, positive response, example #9

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportFirstTestFailedDTC	0B	RFCDTC
#3	DTCStatusAvailabilityMask	FF	DTCSAM

10.2.5.10 Example #10 - ReadDTCInformation, sub-function = reportNumberOfDTCBySeverityMaskRecord

This example demonstrates the usage of the reportNumberOfDTCBySeverityMaskRecord sub-function parameter.

Assumptions:

a) The server supports the testFailed and confirmedDTC status bits for masking purposes.

b) The server supports a total of 3 DTCs which have the following states at the time of the client request.

1) DTC = 112233 hex, StatusOfDTC = 0000 0001 binary, DTCSeverity = 010x xxxx binary = 40 hex, DTCFunctionalUnit = 10 hex:

NOTE Only bit 7 to 5 of the severity byte are valid.

i) Bit 0: testFailed = 1,

ii) Bit 1: testFailedThisMonitoringCycle = n.a.,

iii) Bit 2: pendingDTC = n.a.,

iv) Bit 3: confirmedDTC = 0,

v) Bit 4: testNotCompletedSinceLastClear = n.a.,

vi) Bit 5: testFailedSinceLastClear = n.a.,

vii) Bit 6: testNotCompletedThisMonitoringCycle = n.a.,

viii) Bit 7: warningIndicatorRequested = n.a.

2) DTC 111222 hex, StatusOfDTC = 0000 0001 binary, DTCSeverity = 001x xxxx binary = 20 hex, DTCFunctionalUnit = 10 hex:

NOTE Only bit 7 to 5 of the severity byte are valid.

i) Bit 0: testFailed = 1,

ii) Bit 1: testFailedThisMonitoringCycle = n.a.,

iii) Bit 2: pendingDTC = n.a.,

iv) Bit 3: confirmedDTC = 0,

v) Bit 4: testNotCompletedSinceLastClear = n.a.,

vi) Bit 5: testFailedSinceLastClear = n.a.,

vii) Bit 6: testNotCompletedThisMonitoringCycle = n.a.,

viii) Bit 7: warningIndicatorRequested = n.a.

3) DTC 222333 hex, StatusOfDTC = 0000 1001 binary, DTCSeverity = 100x xxxx binary = 80 hex, DTCFunctionalUnit = 10 hex:

NOTE Only bit 7 to 5 of the severity byte are valid.

i) Bit 0: testFailed = 1,

ii) Bit 1: testFailedThisMonitoringCycle = n.a.,

iii) Bit 2: pendingDTC = n.a.,

iv) Bit 3: confirmedDTC = 1,

v) Bit 4: testNotCompletedSinceLastClear = n.a.,

vi) Bit 5: testFailedSinceLastClear = n.a.,

vii) Bit 6: testNotCompletedThisMonitoringCycle = n.a.,

viii) Bit 7: warningIndicatorRequested = n.a.

In the following example, a count of 2 is returned to the client because DTC 112233 hex and DTC 222333 hex match the client defined severity mask record of C0 01 hex (DTCSeverityMask = 110x xxxx binary = C0 hex, DTCStatusMask = 0000 0001 binary).

NOTE Only bit 7 to 5 of the severity byte are valid.

Table 274 — ReadDTCInformation, sub-function = reportNumberOfDTCBySeverityMaskRecord, request message flow example #10

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportNumberOfDTCBySeverityMaskRecord, responseRequired = yes	07	RNODTCBSMR
#3	DTCSeverityMaskRecord(DTCSeverityMask)	C0	DTCSM
#4	DTCSeverityMaskRecord(DTCStatusMask)	01	DTCSM

Table 275 — ReadDTCInformation, sub-function = reportNumberOfDTCBySeverityMaskRecord, positive response, example #10

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportNumberOfDTCBySeverityMaskRecord	07	RNODTCBSMR
#3	DTCStatusAvailabilityMask	09	DTCSAM
#4	DTCCount [DTCCountHighByte]	00	DTCCHB
#5	DTCCount [DTCCountLowByte]	02	DTCCLB

10.2.5.11 Example #11 - ReadDTCInformation, sub-function = reportDTCDTCBySeverityMaskRecord

This example demonstrates the usage of the reportDTCBySeverityMaskRecord sub-function parameter.

Assumptions: Refer to example #10 (see 10.2.5.10).

In the following example, the DTC 112233hex and DTC 222333 hex which match the client defined severity mask record of C0 01 hex (DTCSeverityMask = 110x xxxx binary = C0 hex, DTCStatusMask = 0000 0001 binary) are reported to the client.

NOTE Only bit 7 to 5 of the severity mask byte are valid.

Table 276 — ReadDTCInformation, sub-function = reportDTCBySeverityMaskRecord, request message flow example #11

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportDTCBySeverityMaskRecord, responseRequired = yes	08	RDTCBSMR
#3	DTCSeverityMaskRecord(DTCSeverityMask)	C0	DTCSM
#4	DTCSeverityMaskRecord(DTCStatusMask)	01	DTCSM

Table 277 — ReadDTCInformation, sub-function = reportDTCBySeverityMaskRecord, positive response, example #11

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportDTCBySeverityMaskRecord	08	RDTCBSMR
#3	DTCStatusAvailabilityMask	09	DTCSAM
#4	DTCSeverityRecord#1(DTCSeverity)	40	DTCS
#5	DTCSeverityRecord#1(DTCFunctionalUnit)	10	DTCFU
#6	DTCSeverityRecord#1(DTCHighByte)	11	DTCHB
#7	DTCSeverityRecord#1(DTCLowByte)	22	DTCLB
#8	DTCSeverityRecord#1(DTCFailureTypeByte)	33	DTCFT
#9	DTCSeverityRecord#1(statusOfDTC)	01	SODTC
#4	DTCSeverityRecord#2(DTCSeverity)	80	DTCS
#5	DTCSeverityRecord#2(DTCFunctionalUnit)	10	DTCFU
#6	DTCSeverityRecord#2(DTCHighByte)	22	DTCHB
#7	DTCSeverityRecord#2(DTCLowByte)	23	DTCLB
#8	DTCSeverityRecord#2(DTCFailureTypeByte)	33	DTCFT
#9	DTCSeverityRecord#2(statusOfDTC)	09	SODTC

10.2.5.12 Example #12 - ReadDTCInformation, sub-function = reportSeverityInformationOfDTC

This example demonstrates the usage of the reportSeverityInformationOfDTC sub-function parameter.

Assumptions: Refer to example #10 (see 10.2.5.10).

In the following example, the DTC 112233 hex, which matches the client defined DTC mask record is reported to the client.

Table 278 — ReadDTCInformation, sub-function = reportSeverityInformationOfDTC, request message flow example #12

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportSeverityInformationOfDTC, responseRequired = yes	09	RSIODTC
#3	DTCMaskRecord [DTCHighByte]	11	DTCHB
#4	DTCMaskRecord [DTCLowByte]	22	DTCLB
#5	DTCMaskRecord [DTCFailureTypeByte]	33	DTCFT

Table 279 — ReadDTCInformation, sub-function = reportSeverityInformationOfDTC, positive response, example #12

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportDTCBySeverityMaskRecord	09	RSIODTC
#3	DTCStatusAvailabilityMask	09	DTCSAM
#4	DTCSeverityRecord[DTCSeverity]	40	DTCS
#5	DTCSeverityRecord[DTCFunctionalUnit]	10	DTCFU
#6	DTCSeverityRecord[DTCHighByte]	11	DTCHB
#7	DTCSeverityRecord[DTCLowByte]	22	DTCLB
#8	DTCSeverityRecord[DTCFailureTypeByte]	33	DTCFT
#9	DTCSeverityRecord[statusOfDTC]	01	SODTC

10.2.5.13 Example #13 - ReadDTCInformation, sub-function = reportDTCExtendedDataRecordByDTCNumber

This example demonstrates the usage of the reportDTCExtendedDataRecordByDTCNumber sub-function parameter.

Assumptions:

a) The server supports the ability to store 2 DTCExtendedData records for a given DTC.

b) Assume that the server requests all available DTCExtendedData records stored by the server for DTC number 123456 hex.

c) Assume that DTC 123456 hex has a statusOfDTC of 24 hex, and that the following extended data is available for the DTC.

d) The DTCExtendedData is referenced via the DTCExtendedDataRecordNumbers 05 hex and 10 hex

Table 280 — DTCExtendedDataRecordNumber 05 hex content

Data Byte	**DTCExtendedDataRecord Contents for DTCExtendedDataRecordNumber 05 hex**	**Byte Value (Hex)**
#1	Warm-up Cycle Counter – Number of warm up cycles since the DTC commanded the MIL to switch off	17

Table 281 — DTCExtendedDataRecordNumber 10 hex content

Data Byte	**DTCExtendedDataRecord Contents for DTCExtendedDataRecordNumber 10 hex**	**Byte Value (Hex)**
#1	DTC Fault Detection Counter – Increments each time the DTC test detects a fault, Decrements each time the test reports no fault.	79

Table 282 — ReadDTCInformation, sub-function = reportDTCExtendedDataRecordByDTCNumber, request message flow example #13

Message direction:	client → server		
Message Type:	Request		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation request SID	19	RDI
#2	sub-function = reportDTCExtendedDataRecordByDTCNumber, responseRequired = yes	06	RDTCEDRBDN
#3	DTCMaskRecord [DTCHighByte]	12	DTCHB
#4	DTCMaskRecord [DTCMiddleByte]	34	DTCMB
#5	DTCMaskRecord [DTCLowByte]	56	DTCLB
#6	DTCEXtendedDataRecordNumber	FF	DTCEDRN

Table 283 — ReadDTCInformation, sub-function = reportDTCExtendedDataRecordByDTCNumber, positive response, example #13

Message direction:	server → client		
Message Type:	Response		
A_Data Byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDTCInformation response SID	59	RDIPR
#2	reportType = reportDTCExtendedDataRecordByDTCNumber	06	RDTCEDRBDN
#3	DTCAndStatusRecord [DTCHighByte]	12	DTCHB
#4	DTCAndStatusRecord [DTCMiddleByte]	34	DTCMB
#5	DTCAndStatusRecord [DTCLowByte]	56	DTCLB
#6	DTCAndStatusRecord [statusOfDTC]	24	SODTC
#7	DTCExtendedDataRecordNumber	05	DTCEDRN
#8	DTCExtendedDataRecord [byte #1]	17	ED_1
#9	DTCExtendedDataRecordNumber	10	DTCEDRN
#10	DTCExtendedDataRecord [byte #1]	79	ED_1

11 InputOutput Control functional unit

Table 284 — InputOutput Control functional unit

Service	Description
InputOutputControlByIdentifier	The client requests the control of an input/output specific to the server.

11.1 InputOutputControlByIdentifier (2F hex) service

The InputOutputControlByIdentifier service is used by the client to substitute a value for an input signal, internal server function and/or control an output (actuator) of an electronic system.

11.1.1 Service description

The client request message contains a inputOutputIdentifier to reference the input signal, internal server function, and/or output signal(s) (actuator(s)) (in case of a device control access it might reference a group of signals) of the server. The controlOptionRecord parameter shall include all information required by the server's input signal(s), internal function(s) and/or output signal(s). Optionally the request message can contain a controlEnableMask, which might be present in case the controlState#1 is used as an inputOutputControlParameter and the inputOutputidentifier to be controlled references more than one parameter (i.e., the inputOutputIdentifier is packeted or bitmapped).

The server shall send a positive response message if the request message was successfully executed. The controlOptionRecord parameter of the requst message can be implemented as a single ON/OFF parameter or as a more complex sequence of control parameters including a number of cycles, a duration, etc. if required.

The service allows the control of multiple inputOutputIdentifiers with their corresponding controlOptionRecord in a single request message. Doing so the server will respond with a single response message including the inputOutputIdentifiers of the request message plus optional controlStatus information.

11.1.2 Request message definition

Table 285 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**InputOutputControlByIdentifier Request Service Id**	**M**	**2F**	**IOCBI**
	inputOutputIdentifier#1[] = [			IOI_
#2	byte 1 (MSB)	M	00-FF	B1
#3	byte 2 (LSB)]	M	00-FF	B2
	controlOptionRecord#1[] = [			CSR_
#4	controlState#1/inputOutputControlParameter	M_1	00-FF	IOCP_/CS_
:	:	:	:	:
#4+(m-1)	controlState#m]	C_1	00-FF	CS_
	controlEnableMaskRecord#1[] = [			CEM_
#4+m	controlMask#1	C_2	00-FF	CM_
:	:	:	:	:
#4+m+(r-1)	controlMask#r]	C_2	00-FF	CM_
:	:	:	:	:
	inputOutputIdentifier#k[] = [			IOI_
#p-q-(n-1)-2	byte 1 (MSB)	U	00-FF	B1
#p-q-(n-1)-1	byte 2 (LSB)]	U	00-FF	B2
	controlOptionRecord#k[] = [			CSR_
#p-q-(n-1)	controlState#1/inputOutputControlParameter	U_1	00-FF	IOCP_/CS_
:	:	:	:	:
#p-q	controlState#n]	U	00-FF	CS_
	controlEnableMaskRecord#k[] = [			CEM_
#p-(q-1)	controlMask#1	C_2	00-FF	CM_
:	:	:	:	:
#p	controlMask#q]	C_2	00-FF	CM_

M_1/U_1 Mandatory: ControlState#1 can be used as either an InputOutputControlParameter or as an additional controlState. If it is used as an InputOutputControlParameter then it shall be implemented as defined in annex E.1.

C_1: The presence of this parameter depends on the inputOutputIdentifier#1-#k and the inputOutputControlParameter of controlOptionRecord#1-#k (if controlState#1 of controlOptionRecord#1-#k is used as an inputOutputControlParameter).

C_2: The presence of this parameter depends on the inputOutputIdentifier#1-#k.

11.1.2.1 Request message sub-function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

11.1.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 286 — Request message data parameter definition

Definition
inputOutputIdentifier
This parameter identifies an server local input signal(s), internal parameter(s) or output signal(s). The applicable range of values for this parameter can be found in the table of recordDataIdentifiers defined in annex C.1.
controlOptionRecord
The controlOptionRecord of each inputOutputIdentifier consists of one or multiple bytes (controlState#1/inputOutputControlParameter to controlState #m). ControlState#1 can be used as either an InputOutputControlParameter that describes how the server has to control its inputs or outputs, or as an additional controlState byte. If it is used as an InputOutputControlParameter then it shall be implemented as defined in annex E.1.

Table 286 — Request message data parameter definition

Definition
controlEnableMaskRecord The ControlEnableMask of each inputOutputIdentifier consists of one or multiple bytes (controlMask#1 to controlMask#r). The ControlEnableMask shall only be supported when the inputOutputControlParameter is used and the inputOutputIdentifier to be controlled consists of more than one parameter (i.e., the inputOutputIdentifier is bit-mapped or packeted by definition). There shall be one bit in the ControlEnableMask corresponding to each individual parameter defined within the inputOutputIdentifier (Note: the parameter could be any number of bits.). The value of each bit shall determine whether the corresponding parameter in the inputOutputIdentifier will be affected by the request. A bit value of '0' in the ControlEnableMask shall represent that the corresponding parameter is not affected by this request and a bit value of '1' shall represent that the corresponding parameter is affected by this request. The most significant bit of ControlMask#1 shall correspond to the first parameter in the ControlState starting at the most significant bit of ControlState#1, the second most significant bit of ControlMask#1 shall correspond to the second parameter in the ControlState, and continuing on in this fashion utilizing as many ControlMask bytes as necessary to mask all parameters. For example, the least significant bit of ControlMask#2 would correspond to the 16th parameter in the controlState. For bitmapped inputOutputIdentifiers, unsupported bits shall also have a corresponding bit in the ControlEnableMask so that the position of the mask bit of every parameter in the ControlEnableMask shall exactly match the position of the corresponding parameter in the controlState.

11.1.3 Positive response message definition

Table 287 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**InputOutputControlByIdentifier Response Service Id**	**S**	**6F**	**IOCBIPR**
#2 #3	inputOutputIdentifier#1[] = [byte 1 (MSB) byte 2 (LSB)]	M M	00-FF 00-FF	IOI_ B1 B2
#4 : #4+(m-1)	controlStatusRecord#1[] = [controlState#1/inputOutputControlParameter : controlState#m]	C_1 : C_2	00-FF : 00-FF	CSR_ IOCP_/CS_ : CS_
:	:	:	:	:
#p-(n-1)-2 #p-(n-1)-1	inputOutputIdentifier#k[] = [byte 1 (MSB) byte 2 (LSB)]	U U	00-FF 00-FF	IOI_ B1 B2
#p-(n-1) : #p	controlStatusRecord#k[] = [controlState#1/inputOutputControlParameter : controlState#n]	C_1 : C_2	00-FF : 00-FF	CSR_ IOCP_/CS_ : CS_

C_1: The presence of this parameter depends on its usage in the request message. ControlState#1 is either used as an InputOutputControlParameter or as an additional controlState. If it is used as an InputOutputControlParameter then it shall be present in the response message and shall be the echo of the InputOutputControlParameter value given in the request message. In all other cases it is user optional to be present (depends on the usage of a controlStatusRecord).

C_2: The presence of this parameter depends on the inputOutputIdentifier and the inputOutputControlParameter (if controlState#1 is used as an inputOutputControlParameter).

11.1.3.1 Positive response message data parameter definition

Table 288 — Response message data parameter definition

Definition
inputOutputIdentifier This parameter is an echo of the inputOutputIdentifier(s) from the request message.
controlStatusRecord The controlState parameter of each inputOutputIdentifier consists of one or multiple bytes (controlState#1/InputOutputControlParameter to controlState #m) which include e.g. feedback data. If controlState#1 was used as an InputOutputControlParameter in the request message than the controlState#1 in the response is the echo of the InputOutputControlParameter value given in the request message (see annex E.1 for details on the InputOutputControlParameter).

11.1.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 289 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This code shall be returnedif the criteria for the request InputOutputControl are not met.	M	CNC
31	**requestOutOfRange** This code shall be returned if: 1. one or multiple of the requested inputOutputIdentifier value(s) is (are) not supported by the device, 2. the client exceeded the maximum number of inputOutputIdentifier allowed to be requested at a time, 3. the client specified a particular inputOutputIdentifier multiple times within a single request message, 4. the inputOutputIdentifier uses the controlState#1 parameter as an inputOutputControlParameter and the value contained in this parameter is invalid (see definition of inputOutputControlParameter), 5. the one or multiple of the applicable controlStates of the controlOptionRecord record are invalid.	M	ROOR
33	**securityAccessDenied** This code shall be returned if a client sends a request with a valid secure inputOutputIdentifier and the server's security feature is currently active.	M	SAD

11.1.5 Message flow example(s) InputOutputControlByIdentifier

The examples below show how the InputOutputControlByIdentifier is used with a Powertrain Control Module (PCM/ECM). All of the examples assume that physical communication is performed with a single server.

11.1.5.1 Example #1 - "Desired Idle Adjustment" resetToDefault

This example uses the controlState#1 parameter of the controlOptionRecord of the request message as an inputOutputControlParameter, therefore the value is echoed back in the response message.

This section specifies the test conditions of the resetToDefault function and the associated message flow of the "Desired Idle Adjustment" inputOutputIdentifier (0132 hex).

Test conditions: ignition=ON, engine at idle speed, engine at operating temperature, vehicle speed=0 [kph]

Conversion: Desired Idle Adjustment [R/MIN] = decimal(Hex) * 10 [r/min]

Table 290 — InputOutputControlByIdentifier request message flow example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier request SID	2F	IOCBI
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 32 (“Desired Idle Adjustment”)	32	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = resetToDefault	01	IOCP_RTD

Table 291 — InputOutputControlByIdentifier positive response message flow example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier response SID	6F	IOCBIPR
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 32 (“Desired Idle Adjustment”)	32	IOI_B2
#4	controlStatusRecord[inputOutputControlParameter] = resetToDefault	01	IOCP_RTD
#5	controlStatusRecord[controlState#1] = 750 r/min	4B	CS_1

11.1.5.2 Example #2 - ”Desired Idle Adjustment” shortTermAdjustment

This example uses the controlState#1 parameter of the controlOptionRecord of the request message as an inputOutputControlParameter, therefore the value is echoed back in the response message.

This section specifies the test conditions of a shortTermAdjustment function and the associated message flow of the "Desired Idle Adjustment" inputOutputIdentifier.

Test conditions: ignition=ON, engine at idle speed, engine at operating temperature, vehicle speed=0 [kph]

Conversion: Desired Idle Adjustment [r/min] = decimal(Hex) * 10 [r/min]

11.1.5.2.1 Step #1: freezeCurrentState

Table 292 — InputOutputControlByIdentifier request message flow example #2 - step #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier request SID	2F	IOCBI
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 32 (“Desired Idle Adjustment”)	32	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = freezeCurrentState	02	IOCP_FCS

Table 293 — InputOutputControlByIdentifier positive response message flow example #2 - step #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier response SID	6F	IOCBIPR
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 32 (“Desired Idle Adjustment”)	32	IOI_B2
#4	controlStatusRecord[inputOutputControlParameter] = freezeCurrentState	02	IOCP_FCS
#5	controlStatusRecord[controlState#1] = 800 r/min]	50	CS_1

11.1.5.2.2 Step #2: shortTermAdjustment

Table 294 — InputOutputControlByIdentifier request message flow example #2 - step #2

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier request SID	2F	IOCBI
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 32 (“Desired Idle Adjustment”)	32	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = shortTermAdjustment	03	IOCP_STA
#5	controlOptionRecord[controlState#1] = 1000 r/min]	64	CS_1

Table 295 — InputOutputControlByIdentifier positive response message flow example #2 - step #2

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier response SID	6F	IOCBLIPR
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 32 (“Desired Idle Adjustment”)	32	IOI_B2
#4	controlStatusRecord[inputOutputControlParameter] = shortTermAdjustment	03	IOCP_STA
#5	controlStatusRecord[controlState#1] = 820 r/min	52	CS_1

NOTE The client has sent an inputOutputControlByIdentifier request message as specified above. The server has sent an immediate positive response message, which includes the controlState parameter "Engine Speed" with the value of "820 r/min". The engine requires a certain amount of time to adjust the idle speed to the requested value of "1000 r/min".

11.1.5.2.3 Step #3: ReadDataByIdentifier

For the example it is assume, that the dataIdentifier F101 hex contains the engine speed parameter.

Table 296 — ReadDataByIdentifier request message flow example #2 - step #3

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier request SID	22	RDBI
#2	recordIdentifier[byte 1] = F1	F1	RI_B1
#3	recordIdentifier[byte 2] = 01	01	RI_B2

Table 297 — ReadDataByIdentifier positive response message flow example #2 - step #3

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	ReadDataByIdentifier response SID	62	RDBIPR
#2	recordIdentifier[byte 1] = F1	F1	RI_B1
#3	recordIdentifier[byte 2] = 01	01	RI_B2
#4	recordValue#1	xx	RV_
:	:	:	:
#m	recordValue#n	xx	RV_

11.1.5.2.4 Step #4: returnControlToECU

Table 298 — InputOutputControlByIdentifier request message flow example #2 - step #4

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier request SID	2F	IOCBI
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 32 (“Desired Idle Adjustment”)	32	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = returnControlToECU	00	RCTECU

Table 299 — InputOutputControlByIdentifier positive response message flow example #2 - step #4

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier response SID	6F	IOCBIPR
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 32 (“Desired Idle Adjustment”)	32	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = returnControlToECU	00	RCTECU

11.1.5.3 Example #3 – EGR and IAC shortTermAdjustment

This example uses a packeted inputOutputIdentifier $0155 to control the EGR and IAC parameters individually or together. The controlState#1 parameter of the controlOptionRecord of the request message is used as an inputOutputControlParameter, therefore the value is echoed back in the response message.

This section specifies the test conditions of a shortTermAdjustment function and the associated message flow of the EGR and IAC inputOutputIdentifier.

Table 300 — inputOutputIdentifier $0155 (IAC / EGR) Data Definition

Position	**Description**
Byte #1	IAC Pintle Position (n = counts)
Byte #2	EGR Duty Cycle: Linear Scaling, 0 counts = 0%, 255 counts = 100%

11.1.5.3.1 Case #1: Control IAC Pintle Position Only

Table 301 — InputOutputControlByIdentifier request message flow example #3 – Case #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier request SID	2F	IOCBI
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 55 (IAC / EGR)	55	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = shortTermAdjustment	03	IOCP_STA
#5	controlOptionRecord[controlState#1] = IAC Pintle Position (7 counts)	07	CS_1
#6	controlOptionRecord[controlState#2] = EGR Duty Cycle (XX%)	XX	CS_2
#7	ControlEnableMask[controlMask#1] = Control IAC Pintle Position ONLY	80	CM_1

NOTE The value transmitted for the EGR Duty Cycle in controlState#2 is irrelevant because the controlMask#1 parameter specifies that only the first parameter in the inputOuputControlIdentifier will be affected by the request.

Table 302 — InputOutputControlByIdentifier positive response message flow example #3 – Case #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier response SID	6F	IOCBLIPR
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 55 (IAC / EGR)	55	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = shortTermAdjustment	03	IOCP_STA
#5	controlOptionRecord[controlState#1] = IAC Pintle Position (7 counts)	07	CS_1
#6	controlOptionRecord[controlState#2] = EGR Duty Cycle (35%)	59	CS_2

11.1.5.3.2 Case #2: Control EGR Duty Cycle Only

Table 303 — InputOutputControlByIdentifier request message flow example #3 – Case #2

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier request SID	2F	IOCBI
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 55 (IAC / EGR)	55	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = shortTermAdjustment	03	IOCP_STA
#5	controlOptionRecord[controlState#1] = IAC Pintle Position (XX counts)	XX	CS_1
#6	controlOptionRecord[controlState#2] = EGR Duty Cycle (45%)	72	CS_2
#7	ControlEnableMask[controlMask#1] = Control EGR Duty Cycle ONLY	40	CM_1

NOTE The value transmitted for the IAC Pintle Position in controlState#1 is irrelevant because the controlMask#1 parameter specifies that only the second parameter in the inputOuputControlIdentifier will be affected by the request.

Table 304 — InputOutputControlByIdentifier positive responsemessage flow example #3 – Case #2

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier response SID	6F	IOCBLIPR
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 55 (IAC / EGR)	55	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = shortTermAdjustment	03	IOCP_STA
#5	controlOptionRecord[controlState#1] = IAC Pintle Position (9 counts)	09	CS_1
#6	controlOptionRecord[controlState#2] = EGR Duty Cycle (45%)	72	CS_2

11.1.5.3.3 Case #3: Control IAC Pintle Position and EGR Duty Cycle

Table 305 — InputOutputControlByIdentifier request message flow example #3 – Case #2

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier request SID	2F	IOCBI
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 55 (IAC / EGR)	55	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = shortTermAdjustment	03	IOCP_STA
#5	controlOptionRecord[controlState#1] = IAC Pintle Position (7 counts)	07	CS_1
#6	controlOptionRecord[controlState#2] = EGR Duty Cycle (45%)	72	CS_2
#7	ControlEnableMask[controlMask#1] = Control IAC and EGR	C0	CM_1

Table 306 — InputOutputControlByIdentifier positive responsemessage flow example #3 – Case #2

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier response SID	6F	IOCBLIPR
#2	inputOutputControlIdentifier[byte 1] = 01	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 55 (IAC / EGR)	55	IOI_B2
#4	controlOptionRecord[inputOutputControlParameter] = shortTermAdjustment	03	IOCP_STA
#5	controlOptionRecord[controlState#1] = IAC Pintle Position (7 counts)	07	CS_1
#6	controlOptionRecord[controlState#2] = EGR Duty Cycle (45%)	72	CS_2

11.1.5.4 Example #4 - Device Control (EGR & IAC Control)

This example uses the controlState#1 parameter of the controlOptionRecord of the request message as an additional control byte.

This message flow example will show how a client could send device control equivalent messages to a server to control multiple inputs/outputs at the same time.

The output control mapping is based on the enable/control byte definitions in the tables below and the brief descriptions that follow:

Table 307 — Example Data Definition

Enable Byte	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
#1	-	-	-	-	-	EGR Enable	IAC 0 = POS; 1 = RPM	IAC Control Enable
Control Byte	**Bit7**	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
#1	IAC Pintle Position (n = counts) or Desired Engine RPM (RPM = n * 12.5)							
#2	EGR Duty Cycle: Linear Scaling, 0 counts = 0%, 255 counts = 100%							

The above given record of enable/control bytes allows the client to

— take control of the Idle Air Control (IAC) motor by placing a 1 in bit 0 of the enable byte,

— command a pintle position or desired engine idle speed (based on the value of the enable byte bit 1) by placing the appropriate value in the first control byte,

— take control of Exhaust Gas Recirculation (EGR) Valve by placing a 1 in bit 2 of the enable byte.

The unused bits/bytes are ignored for the purposes of the examples in this section.

In order to maximize the amount of user data that can be placed in a single request message it is assumed - for this example - that the above shown Enable Byte represents the low byte of the inputOutputControlIdentifier. The high byte of the inputOutputControlIdentifier would be interpreted as a command parameter identifier (CPID) and would be set to 01 hex for this example (can be any value between 00 hex and FF hex).

The above given interpretation of the inputOutputControlIdentifier ends up in the following list of inputOutputControlIdentifier values and their corresponding usage:

Table 308 — inputOutputControlIdentifier values

inputOutputControlIdentifier value (hex)		**Description**	
high byte (CPID)	low byte (enable byte)	resulting value (hex)	
01	**00**	**0100**	Disable IAC Control and EGR Control
01	**01**	**0101**	Control IAC pintle position and disable EGR Control
01	**02**	**0102**	Disable IAC Control and EGR Control
01	**03**	**0103**	Control IAC desired engine RPM and disable EGR control
01	**04**	**0104**	Control EGR Duty Cycle and disable IAC Control
01	**05**	**0105**	Control EGR Duty Cycle and IAC pintle position
01	**06**	**0106**	Control EGR Duty Cycle and disable IAC Control
01	**07**	**0107**	Control EGR Duty Cycle and IAC desired engine RPM

The following message flow shows how the client controls the EGR duty cycle and the IAC pintle position at the same time (single request).

Table 309 — InputOutputControlByIdentifier request message flow example #4 Control EGR Duty Cycle and IAC pintle position

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier request SID	2F	IOCBI
#2	inputOutputControlIdentifier[byte 1] = 01 (CPID)	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 05	05	IOI_B2
#4	controlOptionRecord[controlState#1] = IAC Pintle Position (7 counts)	07	CS_1
#5	controlOptionRecord[controlState#2] = EGR Duty Cycle (35 %)	35	CS_2

Table 310 — InputOutputControlByIdentifier positive response message flow example #4 Control EGR Duty Cycle and IAC pintle position

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier response SID	6F	IOCBIPR
#2	inputOutputControlIdentifier[byte 1] = 01 (CPID)	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 05	05	IOI_B2

The following message flow shows how the client controls the DGR duty cyclew and the IAC desired engine RPM at the same time (single request).

Table 311 — InputOutputControlByIdentifier request message flow example #4 Control EGR Duty Cycle and IAC desired engine RPM

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier request SID	2F	IOCBI
#2	inputOutputControlIdentifier[byte 1] = 01 (CPID)	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 07	07	IOI_B2
#4	controlOptionRecord[controlState#1] = IAC Desired Engine RPM (800)	40	CS_1
#5	controlOptionRecord[controlState#2] = EGR Duty Cycle (43 %)	43	CS_2

Table 312 — InputOutputControlByIdentifier positive response message flow example #4 Control EGR Duty Cycle and IAC desired engine RPM

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	InputOutputControlByIdentifier response SID	6F	IOCBIPR
#2	inputOutputControlIdentifier[byte 1] = 01 (CPID)	01	IOI_B1
#3	inputOutputControlIdentifier[byte 2] = 05	07	IOI_B2

12 Remote Activation of Routine functional unit

Table 313 — Remote Activation of Routine functional unit

Service	Description
RoutineControl	The client requests to start, stop a routine in the server(s) or requests the routine results.

This functional unit specifies the services of remote activation of routines, as they shall be implemented in servers and client. The following section describes two (2) different methods of implementation (Methods "A" and "B"). There may be other methods of implementation possible. Methods "A" and "B" shall be used as a guideline for implementation of routine services.

NOTE Each method may feature the functionality to request routine results service after the routine has been stopped. The selection of method and the implementation is the responsibility of the vehicle manufacturer and system supplier.

The following is a brief description of method "A" and "B":

— **Method "A":**

  — This method is based on the assumption that after a routine has been started by the client in the server's memory the client shall be responsible to stop the routine.

  — The server routine shall be started in the server's memory some time between the completion of the RoutineControl request message that starts the routine and the completion of the first response message (if "positive" based on the server's conditions).

  — The server routine shall be stopped in the server's memory some time after the completion of the StopRoutine request message and the completion of the first response message (if "positive" based on the server's conditions).

  — The client may request routine results after the routine has been stopped.

— **Method "B":**

  — This method is based on the assumption that after a routine has been started by the client in the server's memory that the server shall be responsible to stop the routine.

  — The server routine shall be started in the server's memory some time between the completion of the RoutineControl request message that starts the routine and the completion of the first response message (if "positive" based on the server's conditions).

  — The server routine shall be stopped any time as programmed or previously initialized in the server's memory.

12.1 RoutineControl (31 hex) service

The RoutineControl service is used by the client to:

— start a routine,

— stop a routine, and

— request routine results

referenced by a 2-byte routineIdentifier.

12.1.1 Service description

The following sections specify start routine, stop routine, and request routine results referenced by a routineIdentifier.

The server and the client shall meet the request and response message behaviour as specified in section 6.4 Negative response/confirmation service primitive in case those addressing methods are implemented for this service.

12.1.1.1 Start a routine referenced by a routineIdentifier

The routine shall be started in the server's memory some time between the completion of the StartRoutine request message and the completion of the first response message if the response message is positive or negative, indicating that the request is already performed or in progress to be performed.

The routines could either be tests that run instead of normal operating code or could be routines that are enabled and executed with the normal operating code running. In particular in the first case, it might be necessary to switch the server in a specific diagnostic session using the DiagnosticSessionControl service or to unlock the server using the SecurityAccess service prior to using the StartRoutine service.

12.1.1.2 Stop a routine referenced by a routineIdentifier

The server routine shall be stopped in the server's memory some time after the completion of the StopRoutine request message and the completion of the first response message if the response message is positive or negative, indicating that the request to stop the routine is already performed or in progress to be performed.

NOTE The server routine shall be stopped any time as programmed or previously initialized in the server's memory.

12.1.1.3 Request routine results referenced by a routineIdentifier

This sub-function is used by the client to request results (e.g. exit status information) referenced by a routineIdentifier and generated by the routine which was executed in the server's memory.

Based on the routine results, which may have been received in the positive response message of the stopRoutine sub-function parameter (e.g. normal/abnormalExitWithResults) the requestRoutineResults sub-function shall be used.

An example of routineResults could be data collected by the server, which could not be transmitted during routine execution because of server performance limitations.

12.1.2 Request message definition

Table 314 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**RoutineControl Request Service Id**	**M**	**31**	**RC**
#2	sub-function = [routineControlType]	M	 00-FF	LEV_ RCTP_
 #3 #4	routineIdentifier [] = [byte 1 (MSB) byte 2]	 M M	 00-FF 00-FF	RI_ B1 B2
 #5 : #n	routineControlOptionRecord[] = [routineControlOption#1 : routineControlOption#m]	 C/U : C/U	 00-FF : 00-FF	RCEOR_ RCO_ : RCO _
C: This parameter is user optional to be present for sub-function parameter startRoutine and stopRoutine.				

12.1.2.1 Request message sub-function parameter $Level (LEV_) definition

The sub-function parameters are used by this service to select the control of the routine. Explanations and usage of the possible levels are detailed below (responseRequiredIndicationBit (bit 7) not shown).

Table 315 — Request message sub function definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD
01	**startRoutine** This parameter specifies that the server shall start the routine specified by the routineIdentifier.	U	STR
02	**stopRoutine** This parameter specifies that the server shall stop the routine specified by the routineIdentifier.	U	STPR
03	**requestRoutineResults** This parameter specifies that the server shall return result values of the routine specified by the routineIdentifier.	U	RRR
04 - 7F	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

12.1.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 316 — Request message data parameter definition

Definition
routineIdentifier This parameter identifies a server local routine and is out of the range of defined recordDataIdentifiers (see annex F.1).
routineControlOptionRecord This parameter record contains either — Routine entry option parameters, which optionally specify start conditions of the routine (e.g. timeToRun, startUpVariables, etc.), or — Routine exit option parameters which optionally specify stop conditions of the routine.(e.g. timeToExpireBeforeRoutineStops, variables, etc.).

12.1.3 Positive response message definition

Table 317 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**RoutineControl Response Service Id**	**S**	**71**	**RCPR**
#2	routineControlType	M	00-FF	RCTP_
#3 #4	routineIdentifier [] = [byte 1 (MSB) byte 2]	M M	00-FF 00-FF	RI_ B1 B2
#5 : #n	routineStatusRecord[] = [routineStatus#1 : routineStatus#m]	U : U	00-FF : 00-FF	RSR_ RS_ : RS _

12.1.3.1 Positive response message data parameter definition

Table 318 — Response message data parameter definition

Definition
routineControlType This parameter is an echo of the sub-function parameter from the request message.
routineIdentifier This parameter is an echo of the routineIdentifier from the request message.
routineStatusRecord This parameter record is used to give to the client either — additional information about the status of the server following the start of the routine, — additional information to the client about the status of the server after the routine has been stopped (e.g. totalRunTime, results generated by the routine before stopped, etc., or — results (exit status information) of the routine, which has been stopped previously in the server.

12.1.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 319 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	**subFunctionNotSupported** This code is returned if the requested sub-function is not supported.	M	SFNS
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This code shall be returned if the criteria for the request RoutineControl are not met.	M	CNC
31	**requestOutOfRange** This code shall be returned if: 1. The server does not support the requested routineIdentifier, 2. The user optional routineControlOptionRecord contains invalid data for the requested routineIdentifier.	M	ROOR
33	**securityAccessDenied** This code shall be sent if this code shall be returned if a client sends a request with a valid secure routineIdentifier and the server's security feature is currently active.	M	SAD
72	**GeneralProgrammingFailure** This return code shall be sent if the server detects an error when performing a routine, which accesses server internal memory. An example is when the routine erases or programms a certain memory location in the permanent memory device (e.g. Flash Memory) and the access to that memory location fails.	M	GPF

12.1.5 Message flow example(s) RoutineControl

12.1.5.1 Example #1: sub-function = startRoutine

This section specifies the test conditions to start a routine in the server to continuously test (as fast as possible) all input and output signals on intermittent while a technician would "wiggle" all wiring harness connectors of the system under test. The routineIdentifier references this routine by the routineIdentifier 0201 hex.

Test conditions: ignition = on, engine = off, vehicle speed = 0 [kph]

The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

Table 320 —RoutineControl request message flow - example #1

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RoutineControl request SID	31	RC
#2	sub-function = startRoutine, responseRequired = yes	01	STR
#3	routineIdentifier [byte 1] (MSB)	02	RI_B1
#4	routineIdentifier [byte 2]	01	RI_B2

Table 321 — RoutineControl positive response message flow - example #1

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RoutineControl response SID	71	RCPR
#2	routineControlType = startRoutine	01	STR
#3	routineIdentifier [byte 1] (MSB)	02	RI_B1
#4	routineIdentifier [byte 2]	01	RI_B2

12.1.5.2 Example #2: sub-function = stopRoutine

This section specifies the test conditions to stop a routine in the server which has continuously tested (as fast as possible) all input and output signals on intermittence while a technician would have been "wiggled" all wiring harness connectors of the system under test. The routineIdentifier references this routine by the routineIdentifier 0201 hex.

Test conditions: ignition = on, engine = off, vehicle speed = 0 [kph]

The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

Table 322 — RoutineControl request message flow - example #2

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RoutineControl request SID	31	RC
#2	sub-function = stopRoutine, responseRequired = yes	02	SPR
#3	routineIdentifier [byte 1] (MSB)	02	RI_B1
#4	routineIdentifier [byte 2]	01	RI_B2

Table 323 — RoutineControl positive response message flow - example #2

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	StopRoutine response SID	71	RCPR
#2	routineControlType = stopRoutine	02	SPR
#3	routineIdentifier [byte 1] (MSB)	02	RI_B1
#4	routineIdentifier [byte 2]	01	RI_B2

12.1.5.3 Example #3: sub-function = requestRoutineResults

This section specifies the test conditions to stop a routine in the server which has continuously tested as fast as possible all input and output signals on intermittence while a technician would have been "wiggled" at all wiring harness connectors of the system under test. The routineIdentifier to reference this routine is 0201 hex.

Test conditions: ignition = on, engine = off, vehicle speed = 0 [kph]

The client requests to have a response message by setting the responseRequiredIndicationBit (bit 7 of the sub-function parameter) to '0'.

Table 324 — RequestRoutineResults request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RoutineControl request SID	31	RC
#2	sub-function = requestRoutineResults, responseRequired = yes	03	RRR
#3	routineIdentifier [byte 1] (MSB)	02	RI_B1
#4	routineIdentifier [byte 2]	01	RI_B2

Table 325 — RequestRoutineResults positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RoutineControl response SID	71	RCPR
#2	routineControlType = requestRoutineResults	03	STR
#3	routineIdentifier [byte 1] (MSB)	02	RI_B1
#4	routineIdentifier [byte 2]	01	RI_B2
#5	routineStatusRecord [routineStatus#1] = inputSignal#1	57	RRS_
#6	routineStatusRecord [routineStatus #2] = inputSignal#2	33	RRS_
:	:	:	:
#n	routineStatusRecord [routineStatus #m] = inputSignal#m	8F	RRS_

13 Upload Download functional unit

Table 326 — Upload Download functional unit

Service	Description
RequestDownload	The client requests the negotiation of a data transfer from the client to the server.
RequestUpload	The client requests the negotiation of a data transfer from the server to the client.
TransferData	The client transmits data to the server (download) or requests data from the server (upload).
RequestTransferExit	The client requests the termination of a data transfer.

13.1 RequestDownload (34 hex) service

The requestDownload service is used by the client to initiate a data transfer from the client to the server (download).

13.1.1 Service description

After the server has received the requestDownload request message the server shall take all necessary actions to receive data before it sends a positive response message.

13.1.2 Request message definition

Table 327 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**RequestDownload Request Service Id**	**M**	**34**	**RD**
#2	dataFormatIdentifier	M	00-FF	DFI_
#3	addressAndLengthFormatIdentifier	M	00-FF	ALFID
#4 : #(m-1)+4	memoryAddress[] = [byte 1 (MSB) : byte m]	M : C_1	00-FF : 00-FF	MA_ B1 : Bm
#n-(k-1) : #n	memorySize[] = [byte 1 (MSB) : byte k]	M : C_2	00-FF : 00-FF	MS_ B1 : Bk
C_1: The presence of this parameter depends on address length information parameter of the addressAndLengthFormatIdentifier				
C_2: The presence of this parameter depends on the memory size length information of the addressAndLengthFormatIdentifier.				

13.1.2.1 Request message sub-function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

13.1.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 328 — Request message data parameter definition

Definition
dataFormatIdentifier This data parameter is a one byte value with each nibble encoded separately. The high nibble specifies the "compressionMethod, and the low nibble specifies the "encryptingMethod". The value 00 hex specifies that no compressionMethod nor encryptingMethod is used. Values other than 00 hex are vehicle manufacturer specific.
addressAndLengthFormatIdentifier This parameter is a one byte value with each nibble encoded separately (see annex H.1 for example values): bit 7 - 4: Length (number of bytes) of the memorySize parameter bit 3 - 0: Length (number of bytes) of the memoryAddress parameter
memoryAddress The parameter memoryAddress is the starting address of server memory from which data is to be retrieved. The number of bytes used for this address is defined by the low nibble (bit 3 - 0) of the addressFormatIdentifier. Byte m in the memoryAddress parameter is always the least significant byte of the address being referenced in the server. The most significant byte of the address can be used as a memoryIdentifier. An example of the use of a memoryIdentifier would be a dual processor server with 16 bit addressing and memory address overlap (when a given address is valid for either processor but yields a different physical memory device or internal and external flash is used). In this case, an otherwise unused byte within the memoryAddress parameter can be specified as a memoryIdentifier used to select the desired memory device. Usage of this functionality shall be as defined by vehicle manufacturer / system supplier.
memorySize (unCompressedMemorySize) This parameter shall be used by the server to compare the uncompressed memory size with the total amount of data transferred during the TransferData service. This increases the programming security. The number of bytes used for this size is defined by the high nibble (bit 7 - 4) of the addressFormatIdentifier.

13.1.3 Positive response message definition

Table 329 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**RequestDownload Response Service Id**	**S**	**74**	**RDPR**
#2	lengthFormatIdentifier	M	00-F0	LFID
#3 : #n	maxNumberOfBlockLength = [byte 1 (MSB) : byte m]	M : M	00-FF : 00-FF	MNROB_ B1 : Bm

13.1.3.1 Positive response message data parameter definition

Table 330 — Response message data parameter definition

Definition
lengthFormatIdentifier This parameter is a one byte value with each nibble encoded separately: bit 7 - 4: Length (number of bytes) of the maxNumberOfBlockLength parameter. bit 3 - 0: reserved by document, to be set to 0 hex The format of this parameter is compatible to the format of the addressAndLengthFormatIdentifier parameter contained in the request message, except that the lower nibble has to be set to 0 hex.
maxNumberOfBlockLength This parameter is used by the requestDownload positive response message to inform the client how many data bytes (maxNumberOfBlockLength) shall be included in each TransferData request message from the client. This length reflects the complete message length, including the service identifier and the data parameters present in the TransferData request message. This parameter allows the client to adapt to the receive buffer size of the server before it starts transferring data to the server.

13.1.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 331 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This return code shall be sent if a server receives a request for this service while in the process of receiving a download of a software or calibration module. This could occur if there is a data size mismatch between the server and the client during the download of a module.	M	CNC
31	**requestOutOfRange** This return code shall be sent if 1. The specified dataFormatIdentifier is not valid. 2. The specified addressAndLengthFormatIdentifier is not valid. 3. The specified memoryAddress/memorySize is not valid.	M	ROOR
33	**securityAccessDenied** This return code shall be sent if the server is secure (for server's that support the SecurityAccess service) when a request for this service has been received.	M	SAD
70	**uploadDownloadNotAccepted** This response code indicates that an attempt to download to a server's memory cannot be accomplished due to some fault conditions.	M	UDNA

13.1.5 Message flow example(s) RequestDownload

See section 13.4.5 for a complete message flow example.

13.2 RequestUpload (35 hex) service

The RequestUpload service is used by the client to initiate a data transfer from the server to the client (upload).

13.2.1 Service description

After the server has received the requestUpload request message the server shall take all necessary actions to send data before it sends a positive response message.

13.2.2 Request message definition

Table 332 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**RequestUpload Request Service Id**	**M**	**35**	**RU**
#2	dataFormatIdentifier	M	00-FF	DFI_
#3	addressAndLengthFormatIdentifier	M	00-FF	ALFID
#4 : #(m-1)+4	memoryAddress[] = [byte 1 (MSB) : byte m]	M : C_1	00-FF : 00-FF	MA_ B1 : Bm
#n-(k-1) : #n	memorySize[] = [byte 1 (MSB) : byte k]	M : C_2	00-FF : 00-FF	MS_ B1 : Bk
C_1: The presence of this parameter depends on address length information parameter of the addressAndLengthFormatIdentifier				
C_2: The presence of this parameter depends on the memory size length information of the addressAndLengthFormatIdentifier.				

13.2.2.1 Request message sub-function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

13.2.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 333 — Request message data parameter definition

Definition
dataFormatIdentifier This data parameter is a one byte value with each nibble encoded separately. The high nibble specifies the "compressionMethod, and the low nibble specifies the "encryptingMethod". The value 00 hex specifies that no compressionMethod nor encryptingMethod is used. Values other than 00 hex are vehicle manufacturer specific.
addressAndLengthFormatIdentifier This parameter is a one byte value with each nibble encoded separately (see annex H.1 for example values): bit 7 - 4: Length (number of bytes) of the memorySize parameter bit 3 - 0: Length (number of bytes) of the memoryAddress parameter

Table 333 — Request message data parameter definition

Definition
memoryAddress The parameter memoryAddress is the starting address of server memory from which data is to be retrieved. The number of bytes used for this address is defined by the low nibble (bit 3 - 0) of the addressFormatIdentifier. Byte m in the memoryAddress parameter is always the least significant byte of the address being referenced in the server. The most significant byte of the address can be used as a memoryIdentifier. An example of the use of a memoryIdentifier would be a dual processor server with 16 bit addressing and memory address overlap (when a given address is valid for either processor but yields a different physical memory device or internal and external flash is used). In this case, an otherwise unused byte within the memoryAddress parameter can be specified as a memoryIdentifier used to select the desired memory device. Usage of this functionality shall be as defined by vehicle manufacturer / system supplier.
memorySize (unCompressedMemorySize) This parameter shall be used by the server to compare the uncompressed memory size with the total amount of data transferred during the TransferData service. This increases the programming security. The number of bytes used for this size is defined by the high nibble (bit 7 - 4) of the addressFormatIdentifier.

13.2.3 Positive response message definition

Table 334 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**RequestUpload Response Service Id**	**S**	**75**	**RUPR**
#2	lengthFormatIdentifier	M	00-F0	LFID
#3 : #n	maxNumberOfBlockLength = [byte 1 (MSB) : byte m]	M : M	00-FF : 00-FF	MNROB_ B1 : Bm

13.2.3.1 Positive response message data parameter definition

Table 335 — Response message data parameter definition

Definition
lengthFormatIdentifier This parameter is a one byte value with each nibble encoded separately: bit 7 - 4: Length (number of bytes) of the maxNumberOfBlockLength parameter. bit 3 - 0: reserved by document, to be set to 0 hex The format of this parameter is compatible to the format of the addressAndLengthFormatIdentifier parameter contained in the request message, except that the lower nibble has to be set to 0 hex.
maxNumberOfBlockLength This parameter is used by the requestUpload positive response message to inform the client how many data bytes shall be included in each TransferData positive response message from the server. This length reflects the complete message length, including the service identifier and the data parameters present in the TransferData positve response message.

13.2.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 336 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** This return code shall be sent if a server receives a request for this service while in the process of performing an upload of data. This could occur if there is a data size mismatch between the server and the client during the upload of a module.	M	CNC
31	**requestOutOfRange** This return code shall be sent if 1. The specified dataFormatIdentifier is not valid. 2. The specified addressAndLengthFormatIdentifier is not valid. 3. The specified memoryAddress/memorySize is not valid.	M	ROOR
33	**securityAccessDenied** This return code shall be sent if the server is secure (for server's that support the SecurityAccess service) when a request for this service has been received.	M	SAD
70	**uploadDownloadNotAccepted** This response code indicates that an attempt to upload to a server's memory cannot be accomplished due to some fault conditions.	M	UDNA

13.2.5 Message flow example(s) RequestUpload

See section 13.4.5 for a complete message flow example.

13.3 TransferData (36 hex) service

The TransferData service is used by the client to transfer data either from the client to the server (download) or from the server to the client (upload).

13.3.1 Service description

The data transfer direction is defined by the preceding RequestDownload or RequestUpload service. If the client initiated a RequestDownload the data to be downloaded is included in the parameter(s) transferRequestParameter in the TransferData request message(s). If the client initiated a RequestUpload the data to be uploaded is included in the parameter(s) transferRequestParameter in the TransferData request message(s).

The TransferData service request includes a blockSequenceCounter to allow for an improved error handling in case a TransferData service fails during a sequence of multiple TransferData requests. The blockSequenceCounter of the server shall be initialized to one (1) when receiving a RequestDownload (34 hex) or RequestUpload (35 hex) request message. This means that the first TransferData (36 hex) request message following the RequestDownload (34 hex) or RequestUpload (35 hex) request messge starts with a blockSequenceCounter of one (1).

13.3.2 Request message definition

Table 337 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**TransferData Request Service Id**	**M**	**36**	**TD**
#2	blockSequenceCounter	M	00-FF	BSC
#3 : #n	transferRequestParameterRecord[] = [transferRequestParameter#1 : transferRequestParameter#m]	 U : U	 00-FF : 00-FF	TRPR_ TRTP_ : TRTP_

13.3.2.1 Request message sub-function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

13.3.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 338 — Request message data parameter definition

Definition
blockSequenceCounter The blockSequenceCounter parameter value starts at 01 hex with the first TransferData request that follows the RequestDownload (34 hex) or RequestUpload (35 hex) service. Its value is incremented by 1 for each subsequent TransferData request. At the value of FF hex the blockSequenceCounter rolls over and starts at 00 hex with the next TransferData request message. Example use cases: a) If a TransferData request to download data is correctly received and processed in the server but the positive response message does not reach the client then the client would determine an application layer timeout and would repeat the same request (including the same blockSequenceCounter). The server would receive the repeated TransferData request and could determine based on the included blockSequenceCounter that this TransferData request is repeated. The server would send the positive response message immediately without writing the data once again into its memory. b) If the TransferData request to download data is not received correctly in the server then the server would not send a positive response message. The client would determine an application layer timeout and would repeat the same request (including the same blockSequenceCounter). The server would receive the repeated TransferData request and could determine based on the included blockSequenceCounter that this is a new TransferData. The server would process the service and would send the positive response message. c) If a TransferData request to upload data is correctly received and processed in the server but the positive response message does not reach the client then the client would determine an application layer timeout and would repeat the same request (including the same blockSequenceCounter). The server would receive the repeated TransferData request and could determine based on the included blockSequenceCounter that this TransferData request is repeated. The server would send the positive response message immediately accessing the previously provided data once again in its memory. d) If the TransferData request to upload data is not received correctly in the server then the server would not send a positive response message. The client would determine an application layer timeout and would repeat the same request (including the same blockSequenceCounter). The server would receive the repeated TransferData request and could determine based on the included blockSequenceCounter that this is a new TransferData. The server would process the service and would send the positive response message.
transferRequestParameterRecord This parameter record contains parameter(s) which are required by the server to support the transfer of data. Format and length of this parameter(s) are vehicle manufacturer specific. Examples: For a download, the transferRequestParameterRecord could include the data to be transferred. For an upload, the transferRequestParameterRecord parameter could include the address and number of bytes to retrieve data.

13.3.3 Positive response message definition

Table 339 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**TransferData Response Service Id**	**S**	**76**	**TDPR**
#2	blockSequenceCounter	M	00-FF	BSC
#3 : #n	transferResponseParameterRecord[] = [transferResponseParameter#1 : transferResponseParameter#m]	U : U	00-FF : 00-FF	TREPR_ TREP_ : TREP

13.3.3.1 Positive response message data parameter definition

Table 340 — Response message data parameter definition

Definition
blockSequenceCounter This parameter is an echo of the blockSequenceCounter parameter from the request message.
transferResponseParameterRecord This parameter shall contain parameter(s), which are required by the client to support the transfer of data. Format and length of this parameter(s) are vehicle manufacturer specific. Examples: For a download, the parameter transferResponseParameterRecord could include a checksum computed by the server. For an upload, the parameter transferResponseParameterRecord could include the uploaded data. Note that for a download, the parameter transferResponseParameterRecord should not repeat the transferRequestParameterRecord.

13.3.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 341 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** The RequestDownload or RequestUpload service is not active when a request for this service is received.	M	CNC
31	**requestOutOfRange** This return code shall be sent if the transferRequestParameterRecord contains additional control paremeters (e.g. additional address information) and this control information is invalid.	M	ROOR
71	**transferDataSuspended** This response code indicates that a data transfer operation was halted due to some fault.	M	TDS
72	**generalProgrammingFailure** This return code shall be sent if the server detects an error when erasing or programming a memory location in the permanent memory device (e.g. Flash Memory) during the download of data.	M	GPF

Table 341 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
73	**wrongBlockSequenceCounter** This return code shall be sent if the server detects an error in the sequence of the blockSequenceCounter. Note that the repetition of a TransferData request message with a blockSequenceCounter equal to the one included in the previous TransferData request message shall be accepted by the server.	M	WBSC
8D / 8E	**VoltageTooHigh** / **VoltageTooLow** This return code shall be sent as applicable if the voltage measured at the primary power pin of the server is out of the acceptable range for downloading data into the server's permanent memory (e.g. Flash Memory).	M	VTH / VTL

13.3.5 Message flow example(s) TransferData

See section 13.4.5 for a complete message flow example.

13.4 RequestTransferExit (37 hex) service

This service is used by the client to terminate a data transfer between client and server (upload or download).

13.4.1 Service description

This service is used by the client to terminate a data transfer between client and server (upload or download).

13.4.2 Request message definition

Table 342 — Request message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**RequestTransferExit Request Service Id**	**M**	**37**	**RTE**
#2 : #n	transferRequestParameterRecord[] = [transferRequestParameter#1 : transferRequestParameter#m]	U : U	00-FF : 00-FF	TRPR_ TRTP_ : TRTP_

13.4.2.1 Request message sub-function parameter $Level (LEV_) definition

This service does not use a sub-function parameter.

13.4.2.2 Request message data parameter definition

The following data-parameters are defined for this service:

Table 343 — Request message data parameter definition

Definition
transferRequestParameterRecord This parameter record contains parameter(s), which are required by the server to support the transfer of data. Format and length of this parameter(s) are vehicle manufacturer specific.

13.4.3 Positive response message definition

Table 344 — Positive response message definition

A_Data byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	**RequestTransferExit Response Service Id**	**S**	**77**	**RTEPR**
#2 : #n	transferResponseParameterRecord[] = [transferResponseParameter#1 : transferResponseParameter#m]	U : U	00-FF : 00-FF	TREPR_ TREP_ : TREP

13.4.3.1 Positive response message data parameter definition

Table 345 — Response message data parameter definition

Definition
transferResponseParameterRecord This parameter shall contain parameter(s) which are required by the client to support the transfer of data. Format and length of this parameter(s) are vehicle manufacturer specific.

13.4.4 Supported negative response codes (NRC_)

The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 346 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	**incorrectMessageLengthOrInvalidFormat** The length of the message is wrong	M	IMLOIF
22	**conditionsNotCorrect** The RequestDownload or RequestUpload service is not active or the programming process is not completed when a request for this service is received.	M	CNC

13.4.5 Message flow example(s) for downloading/uploading data

13.4.5.1 Download data to a server

This section specifies the conditions to transfer data (download) from the client to the server.

The example consists of three (3) steps.

In the **1st step** the client and the server execute a requestDownload service. With this service the following information is exchanged as parameters in the request and positive response message between client and the server:

Table 347 — Definition of transferRequestParameter values

Data Parameter Name	Data Parameter Value(s) (Hex)	Data Parameter Description
memoryAddress (3 bytes)	602000	memoryAddress (start) to download data to
dataFormatIdentifier	11	dataFormatIdentifier (compressionMethod = $1x) (encryptingMethod = $x1)
unCompressedMemorySize (3 bytes)	00FFFF	uncompressedMemorySize = (64 Kbytes) This parameter value shall be used by the server to compare to the actual number of bytes transferred during the execution of the requestTransferExit service.

Table 348 — Definition of transferResponseParameter value

Data Parameter Name	Data Parameter Value(s) (Hex)	Data Parameter Description
maximumNumberOfBlockLength	0081	maximumNumberOfBlockLength: (serviceId + BlockSequenceCounter (1 byte) + 127 server data bytes = 129 data bytes)

In the **2nd step** the client transfers 64 KBytes (number of transferData services with 127 data bytes can not be calculated because the compression method and its compression ratio is supplier specific) of data to the flash memory starting at memoryaddress 602000 hex to the server.

In the **3rd step** the client terminates the data transfer to the server with a requestTransferExit service.

Test conditions: ignition = on, engine = off, vehicle speed = 0 [kph]

It is assumed, that for this example the server supports a three (3) byte memoryAddress and a three (3) byte unCompressedMemorySize. Furthermore it is assumed that the server supports a blockSequenceCounter in the TransferData (36 hex) service. The number of TransferData services with 127 data bytes can not be calculated because the compression method and its compression ratio is supplier specific. Therefore is assumed that the last TransferData request message contains a blockSequenceCounter equal to 68 hex.

13.4.5.1.1 Step #1: Request for download

Table 349 — RequestDownload request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RequestDownload request SID	34	RD
#2	dataFormatIdentifier	11	DFI
#3	addressAndLengthFormatIdentifier	33	ALFID
#4	memoryAddress[Byte1] (MSB)	60	MA_B1
#5	memoryAddress[Byte2]	20	MA_B2
#6	memoryAddress[Byte3] (LSB)	00	MA_B3
#7	unCompressedMemorySize[Byte1] (MSB)	00	UCMS_B1
#8	unCompressedMemorySize[Byte2]	FF	UCMS_B2
#9	unCompressedMemorySize[Byte3] (LSB)	FF	UCMS_B3

Table 350 — RequestDownload positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RequestDownload response SID	74	RDPR
#2	LengthFormatIdentifier	20	LFID
#3	maxNumberOfBlockLength [byte #1] (MSB)	00	MNROB_B1
#4	maxNumberOfBlockLength [byte #2] (LSB)	81	MNROB_B1

13.4.5.1.2 Step #2: Transfer data

Table 351 — TransferData request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TransferData request SID	36	TD
#2	blockSequenceCounter	01	BSC
#3	transferRequestParameterRecord[transferRequestParameter#1] = dataByte3	xx	TRTP_1
:	:	:	:
#129	transferRequestParameterRecord[transferRequestParameter#127] = dataByte129	xx	TRTP_127

Table 352 — TransferData positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TransferData response SID	76	TDPR
#2	blockSequenceCounter	01	BSC

:

Table 353 — TransferData request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TransferData request SID	36	TD
#2	blockSequenceCounter	68	BSC
#3	transferRequestParameterRecord[transferRequestParameter#1] = dataByte3	xx	TRTP_1
:	:	:	:
#n+2	transferRequestParameterRecord[transferRequestParameter#n-2] = dataByte n	xx	TRTP_n-2

Table 354 — TransferData positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TransferData response SID	76	TDPR

Table 354 — TransferData positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#2	blockSequenceCounter	68	BSC

13.4.5.1.3 Step #3: Request Transfer exit

Table 355 — RequestTransferExit request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RequestTransferExit request SID	37	RTE

Table 356 — RequestTransferExit positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RequestTransferExit response SID	77	RTEPR

13.4.5.2 Upload data from a server

This section specifies the conditions to transfer data (upload) from a server to the client.

The example consists of three (3) steps.

In the **1st step** the client and the server execute a requestUpload service. With this service the following information is exchanged as parameters in the request and positive response message between client and the server:

Table 357 — Definition of transferRequestParameter values

Data Parameter Name	**Data Parameter Value(s) (Hex)**	**Data Parameter Description**
memoryAddress (3 bytes)	201000	memoryAddress (start) to upload data from
memorySize	11	dataFormatIdentifier (compressionMethod = $1x) (encryptingMethod = $x1)
uncompressedMemorySize (3 bytes)	0001FF	uncompressedMemorySize = (511 bytes) This parameter value shall indicate how many data bytes shall be transferred and shall be used by the server to compare to the actual number of bytes transferred during execution of the requestTransferExit service.

Table 358 — Definition of transferResponseParameter value

Data Parameter Name	Data Parameter Value(s) (Hex)	Data Parameter Description
maximumNumberOfBlockLength	0081	maximumNumberOfBlockLength: (serviceId + 128 server data bytes = 129 data bytes)

In the **2nd step** the client transfers 511 data bytes (4 transferData services with 129 (127 server data bytes + 1 serviceId data byte + 1 blockSequenceCounter byte) data bytes and 1 transferData service with 5 (3 server data bytes + 1 serviceId data byte+ 1 blockSequenceCounter byte) data bytes from the external RAM starting at memoryaddress 201000 hex in the server.

In the **3rd step** the client terminates the data transfer to the server with a requestTransferExit service.

Test conditions: ignition = on, engine = off, vehicle speed = 0 [kph]

It is assumed, that for this example the server supports a three (3) byte memoryAddress and a three (3) byte unCompressedMemorySize. Furthermore it is assumed that the server supports a blockSequenceCounter in the TransferData (36 hex) service. The number of TransferData services with 128 data bytes can not be calculated because the compression method and its compression ratio is supplier specific. Therefore is assumed that the last TransferData request message contains a blockSequenceCounter equal to 68 hex.

13.4.5.2.1 Step #1: Request for upload

Table 359 — RequestUpload request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RequestUpload request SID	35	RU
#2	dataFormatIdentifier	11	DFI
#3	addressAndLengthFormatIdentifier	33	ALFID
#4	memoryAddress[Byte1] (MSB)	20	MA_B1
#5	memoryAddress[Byte2]	10	MA_B2
#6	memoryAddress[Byte3] (LSB)	00	MA_B3
#7	unCompressedMemorySize[Byte1] (MSB)	00	UCMS_B1
#8	unCompressedMemorySize[Byte2]	01	UCMS_B2
#9	unCompressedMemorySize[Byte3] (LSB)	FF	UCMS_B3

Table 360 — RequestUpload positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RequestUpload response SID	75	RUPR
#2	lengthFormatIdentifier	20	LFID
#3	maxNumberOfBlockLength [byte #1] (MSB)	00	MNROB_B1
#4	maxNumberOfBlockLength [byte #2] (LSB)	81	MNROB_B1

13.4.5.2.2 Step #2: Transfer data

Table 361 — TransferData request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TransferData request SID	36	TD
#2	blockSequenceCounter	01	BSC

Table 362 — TransferData positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TransferData response SID	76	TDPR
#2	blockSequenceCounter	01	BSC
#3	transferResponseParameterRecord[transferResponseParameter#1] = dataByte3	xx	TREP_1
.	:	:	:
#129	transferResponseParameterRecord[transferResponseParameter#127] = dataByte129	xx	TREP_127

⋮

Table 363 — TransferData request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TransferData request SID	36	TD
#2	blockSequenceCounter	04	BSC

Table 364 — TransferData positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	TransferData response SID	76	TDPR
#2	blockSequenceCounter	04	BSC
#3	transferResponseParameterRecord[transferResponseParameter#1] = dataByte3	xx	TREP_1
#4	:	:	:
#5	transferResponseParameterRecord[transferResponseParameter#127] = dataByte5	xx	TREP_5

13.4.5.2.3 Step #3: Request Transfer exit

Table 365 —RequestTransferExit request message flow example

Message direction:	client → server		
Message Type:	Request		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RequestTransferExit request SID	37	RTE

Table 366 — RequestTransferExit positive response message flow example

Message direction:	server → client		
Message Type:	Response		
A_Data byte	**Description (all values are in hexadecimal)**	**Byte Value (Hex)**	**Mnemonic**
#1	RequestTransferExit response SID	77	RTEPR

Annex A
(normative)

Global parameter definitions

A.1 Negative response codes

The following table defines all negative response codes used within this standard. Note that each service specifies the negative response codes applicable for the service.

The negative response code range 00 – FF hex is divided into 2 ranges:

— 00 – 7F hex: communication related negative response codes

— 80 – FF hex: negative response codes for specific conditions that are not correct at the point in time the request is received by the server

Table A.1 — Definition of responseCode values

Hex value	responseCode	Mnemonic
00 -10	**reservedByDocument** This range of values is reserved by this document for future definition.	RBD
11	**serviceNotSupported** This response code indicates that the requested action will not be taken because the server does not support the requested service. The server shall send this response code in case the client has sent a request message with a service identifier, which is either unknown or not supported by the server. Therefore this negative response code is not shown in the list of negative response codes to be supported for a diagnostic service, because this negative response code is not applicable for supported services.	SNS
12	**subFunctionNotSupported** This response code indicates that the requested action will not be taken because the server does not support the service specific parameters of the request message. The server shall send this response code in case the client has sent a request message with a known and supported service identifier but with "sub parameters“ which are either unknown or not supported.	SFNS
13	**incorrectMessageLengthOrInvalidFormat** This response code indicates that the requested action will not be taken because the length of the received request message does not match the prescribed length for the specified service or the format of the paramters do not match the prescribed format for the specified service.	IMLOIF
14 - 20	**reservedByDocument** This range of values is reserved by this document for future definition.	RBD

Table A.1 — Definition of responseCode values

Hex value	responseCode	Mnemonic
21	**busyRepeatRequest** This response code indicates that the server is temporarily too busy to perform the requested operation. In this circumstance the client shall perform repetition of the "identical request message" or "another request message". The repetition of the request shall be delayed by a time specified in the respective implementation documents. Example: In a multi-client environment the diagnostic request of one client might be blocked temporarily by a NRC $21 while a different client finishes a diagnostic task. Note: If the server is able to perform the diagnostic task but needs additional time to finish the task and prepare the response, the NRC $78 shall be used instead of NRC $21. This response code is in general supported by each diagnostic service, as not otherwise stated in the data link specific implementation document, therefore it is not listed in the list of applicable response codes of the diagnostic services.	BRR
22	**conditionsNotCorrect** This response code indicates that the requested action will not be taken because the server prerequisite conditions are not met. This may occur when sequence sensitive requests are issued in the wrong order. The server shall send this response code in case the client has sent a known and supported request message at a time where the server has expected another request message because of a predefined sequence of services. A typical example of occurrence is the securityAccess service, which requires a sequence of messages as specified in the message description of this service.	CNC
23	**reservedByDocument** This range of values is reserved by this document for future definition.	RBD
24	**requestSequenceError** This response code indicates that the requested action will not be taken because the server expects a different sequence of request messages or message as send by the client. EXAMPLE A successful SecurityAccess service specifies a sequence of requestSeed and sendKey as susb-fuctions in the request messages. If the sequence is sent different by the client the server shall send a neagtive response message with the negative response code 24 hex - . requestSequenceError.	RSE
25 - 30	**reservedByDocument** This range of values is reserved by this document for future definition.	RBD
31	**requestOutOfRange** This response code indicates that the requested action will not be taken because the server has detected that the request message contains a parameter which attempts to substitute a value beyond its range of authority (e.g. attempting to substitute a data byte of 111 when the data is only defined to 100). The server shall send this response code in if the client has sent a request message including data bytes to adjust a variant, which does not exist (invalid data) in the server. This response code shall be implemented for all services, which allow the client to write data or adjust functions by data in the server.	ROOR
32	**reservedByDocument** This range of values is reserved by this document for future definition.	RBD

Table A.1 — Definition of responseCode values

Hex value	responseCode	Mnemonic
33	**securityAccessDenied** This response code indicates that the requested action will not be taken because the server's security strategy has not been satisfied by the client. The server shall send this response code if one of the following cases occur: - the test conditions of the server are not met, - the required message sequence e.g. DiagnosticSessionControl, securityAccess is not met, - the client has sent a request message which requires an unlocked server. Beside the mandatory use of this negative response code as specified in the applicable services within this standard, this negative response code can also be used for any case where security is required and is not yet granted to perform the required service.	SAD
34	**reservedByDocument** This range of values is reserved by this document for future definition.	RBD
35	**invalidKey** This response code indicates that the server has not given security access because the key sent by the client did not match with the key in the server's memory. This counts as an attempt to gain security. The server shall remain locked and increment ist internal securityAccessFailed counter.	IK
36	**exceedNumberOfAttempts** This response code indicates that the requested action will not be taken because the client has unsuccessfully attempted to gain security access more times than the server's security strategy will allow.	ENOA
37	**requiredTimeDelayNotExpired** This response code indicates that the requested action will not be taken because the client's latest attempt to gain security access was initiated before the server's required timeout period had elapsed.	RTDNE
38 – 4F	**reservedByExtendedDataLinkSecurityDocument** This range of values is reserved by ISO 15764 Extended data link security.	RBEDLSD
50 – 6F	**reservedByDocument** This range of values is reserved by this document for future definition.	RBD
70	**uploadDownloadNotAccepted** This response code indicates that an attempt to upload/download to a server's memory cannot be accomplished due to some fault conditions.	UDNA
71	**transferDataSuspended** This response code indicates that a data transfer operation was halted due to some fault.	TDS
72	**generalProgrammingFailure** This response code indicates that the server detected an error when erasing or programming a memory location in the permanent memory device (e.g. Flash Memory).	GPF
73	**wrongBlockSequenceCounter** This response code indicates that the server detected an error in the sequence of blockSequenceCounter values. Note that the repetition of a TransferData request message with a blockSequenceCounter equal to the one included in the previous TransferData request message shall be accepted by the server.	WBSC
74 - 77	**ReservedByDocument** This range of values is reserved by this document for future definition.	RBD

Table A.1 — Definition of responseCode values

Hex value	responseCode	Mnemonic
78	**requestCorrectlyReceived-ResponsePending** This response code indicates that the request message was received correctly, and that all parameters in the request message were valid, but the action to be performed is not yet completed and the server is not yet ready to receive another request. As soon as the requested service has been completed, the server shall send a positive response message or negative response message with a response code different from this. The negative response message with this response code may be repeated by the server until the requested service is completed and the final response message is sent. This response code might impact the application layer timing parameter values. The detailed specification shall be included in the data link specific implementation document. **This response code shall only be used in a negative response message if the server will not be able to receive further request messages from the client while completing the requested diagnostic service.** A typical example where this response code may be used is when the client has sent a request message, which includes data to be programmed or erased in flash memory of the server. If the programming/erasing routine (usually executed out of RAM) is not able to support serial communication while writing to the flash memory the server shall send a negative response message with this response code. This response code is in general supported by each diagnostic service, as not otherwise stated in the data link specific implementation document, therefore it is not listed in the list of applicable response codes of the diagnostic services.	RCRRP
79 – 7D	**reservedByDocument** This range of values is reserved by this document for future definition.	RBD
7E	**subFunctionNotSupportedInActiveSession** This response code indicates that the requested action will not be taken because the server does not support the requested sub-function in the session currently active. This response code is in general supported by each diagnostic service with a sub-function parameter, as not otherwise stated in the data link specific implementation document, therefore it is not listed in the list of applicable response codes of the diagnostic services.	SFNSIAS
7F	**serviceNotSupportedInActiveSession** This response code indicates that the requested action will not be taken because the server does not support the requested service in the session currently active. This response code is in general supported by each diagnostic service, as not otherwise stated in the data link specific implementation document, therefore it is not listed in the list of applicable response codes of the diagnostic services.	SNSIAS
81	**rpmTooHigh** This response code indicates that the requested action will not be taken because the server prerequisite condition for RPM is not met (current RPM is above a pre-programmed maximum threshold).	RPMTH
82	**rpmTooLow** This response code indicates that the requested action will not be taken because the server prerequisite condition for RPM is not met (current RPM is below a pre-programmed minimum threshold).	RPMTL
83	**engineIsRunning** This is required for those actuator tests which cannot be actuated while the Engine is running. This is different from RPM too high negative response, and needs to be allowed.	EIR
84	**engineIsNotRunning** This is required for those actuator tests which cannot be actuated unless the Engine is running. This is different from RPM too low negative response, and needs to be allowed.	EINR
85	**engineRunTimeTooLow** This response code indicates that the requested action will not be taken because the server prerequisite condition for engine run time is not met (current engine run time is below a pre-programmed limit).	ERTTL

Table A.1 — Definition of responseCode values

Hex value	responseCode	Mnemonic
86	**temperatureTooHigh** This response code indicates that the requested action will not be taken because the server prerequisite condition for temperature is not met (current temperature is above a pre-programmed maximum threshold).	TEMPTH
87	**temperatureTooLow** This response code indicates that the requested action will not be taken because the server prerequisite condition for temperature is not met (current temperature is below a pre-programmed minimum threshold).	TEMPTL
88	**vehicleSpeedTooHigh** This response code indicates that the requested action will not be taken because the server prerequisite condition for vehicle speed is not met (current VS is above a pre-programmed maximum threshold).	VSTH
89	**vehicleSpeedTooLow** This response code indicates that the requested action will not be taken because the server prerequisite condition for vehicle speed is not met (current VS is below a pre-programmed minimum threshold).	VSTL
8A	**throttle/PedalTooHigh** This response code indicates that the requested action will not be taken because the server prerequisite condition for throttle/pedal position is not met (current TP/APP is above a pre-programmed maximum threshold).	TPTH
8B	**throttle/PedalTooLow** This response code indicates that the requested action will not be taken because the server prerequisite condition for throttle/pedal position is not met (current TP/APP is below a pre-programmed minimum threshold).	TPTL
8C	**transmissionRangeInNeutral** This response code indicates that the requested action will not be taken because the server prerequisite condition for being in neutral is not met (current transmission range is not in neutral).	TRIN
8D	**transmissionRangeInGear** This response code indicates that the requested action will not be taken because the server prerequisite condition for being in gear is not met (current transmission range is not in gear).	TRIG
8F	**brakeSwitch(es)NotClosed (Brake Pedal not pressed or not applied)** For saftey reasons, this is required for certain tests before it begins, and must be maintained for the entire duration of the test.	BSNC
90	**shifterLeverNotInPark** For saftey reasons, this is required for certain tests before it begins, and must be maintained for the entire duration of the test.	SLNIP
91	**torqueConverterClutchLocked** This response code indicates that the requested action will not be taken because the server prerequisite condition for torque converter clutch is not met (current TCC status above a pre-programmed limit or locked).	TCCL
92	**voltageTooLow** This response code indicates that the requested action will not be taken because the server prerequisite condition for voltage at the primary pin of the server (ECU) is not met (current voltage is below a pre-programmed maximum threshold).	VTL
93	**voltageTooHigh** This response code indicates that the requested action will not be taken because the server prerequisite condition for voltage at the primary pin of the server (ECU) is not met (current voltage is above a pre-programmed maximum threshold).	VTH
94 - FE	**reservedForSpecificConditionsNotCorrect** This range of values is reserved by this document for future definition.	RFSCNC

Table A.1 — Definition of responseCode values

Hex value	responseCode	Mnemonic
FF	**reservedByDocument** This range of values is reserved by this document for future definition.	RBD

Annex B
(normative)

Diagnostic and communication management functional unit data parameter definitions

B.1 communicationType parameter definition

The communicationType is a 1-byte value. The bits represent the communicationTypes, which can be controlled via the CommunicationControl (28 hex) service. Only the lower 3 bits (bit 0-2) of the 1-byte value are used to represent the communicationType information. The remaining bits (bit 3-7) are reserved for future definition and have to be set to zero (0). For example, a communicationType with a bit combination (Bits 2-0) of "011b" is valid and disables BOTH "normalCommunicationMessages" and "networkManagementCommunicationMessages" messages. Based on this the following bit coding applies.

Table B.1 — Definition of communicationType byte definition

Bit 2-0	Description	Cvt	Mnemonic
001b	**normalCommunicationMessages** This bit references all application-related communication (inter-application signal exchange between multiple in-vehicle servers).	M	NCM
010b	**networkManagementCommunicationMessages** This bit references all network management related communication.	M	NWMCM
100b	**diagnosticCommunicationMessages** This bit references all diagnostic related communication.	M	DIAGCM

B.2 eventWindowTime parameter definition

Table B.2 — Definition of eventWindowTime parameter values

Hex	Description	Cvt	Mnemonic
00 - 01	**reservedByDocument**	M	RBD
	This value is reserved by the document		
02	**infiniteTimeToResponse**	U	ITTR
	This value specifies that the event window shall stay active for an infinite amount of time (e.g. open window until power off).		
03-7F	**vehicleManufacturerSpecific**	U	VMS
	This range of values is reserved for vehicle manufacturer specific use. The resolution of the eventWindowTime parameter is left vehicle manufacturer discretionary.		
80-FF	**reservedByDocument**	M	RBD
	This range of values is reserved by this document for future definition.		

B.3 baudrateIdentifier parameter definition

Table B.3 — Definition of baudrateIdentifier Values

Hex	Description	Cvt	Mnemonic
00	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD
01	**PC9600Baud** This value specifies the standard PC baudrate of 9.6 KBaud.	U	PC9600
02	**PC19200Baud** This value specifies the standard PC baudrate of 19.2 KBaud.	U	PC19200
03	**PC38400Baud** This value specifies the standard PC baudrate of 38.4 KBaud.	U	PC38400
04	**PC57600Baud** This value specifies the standard PC baudrate of 57.6 KBaud.	U	PC57600
05	**PC115200Baud** This value specifies the standard PC baudrate of 115.2 KBaud.	U	PC115200
06 - FF	**reservedByDocument** This range of values is reserved by this document for future definition.	M	RBD

Annex C
(normative)

Data transmission functional unit data parameter definitions

C.1 recordDataIdentifier parameter definitions

The parameter recordDataIdentifier (RDI) is to identify a server specific local data record. This parameter shall be available in the server's memory. The recordDataIdentifier value shall either exist in fixed memory or temporarily stored in RAM if defined dynamically by the service dynamicallyDefineDataIdentifier. Values are defined in the table below.

Table C.1 — recordDataIdentifier data parameter definitions

Hex	Description	Cvt	Mnemonic
0000 - 00FF	**reservedByDocument** This range of values shall be reserved by this document for future definition.	M	RBD
0100 - EFFF	**vehicleManufacturerSpecific** This range of values shall be used to reference vehicle manufacturer specific record data identifiers and input/output identifiers within the server.	U	VMS
F000 - F00F	**networkConfigurationDataForTractorTrailerApplication** This value shall be used to request the remote addresses of all trailer systems independent of their functionality.	U	NCDFTTA
F010 - F0FF	**networkConfigurationData** This range of values is reserved to represent network configuration data.	U	NCD
F100 - F17F	**identificationOption-VehicleManufacturerSpecific** This range of values shall be used for vehicle manufacturer specific server/vehicle identification options.	U	IO-VMS
F180	**bootSoftwareIdentification** This value shall be used to reference the vehicle manufacturer specific ECU boot software identification record. The first data byte of the record data shall be the numberOfModules that are reported. Following the numberOfModules the boot software identification(s) are reported. The format of the boot software identification structure shall be ECU specific and defined by the vehicle manufacturer.	U	BSI
F181	**applicationSoftwareIdentification** This value shall be used to reference the vehicle manufacturer specific ECU application software number(s). The first data byte of the record data shall be the numberOfModules that are reported. Following the numberOfModules the application software identification(s) are reported. The format of the application software identification structure shall be ECU specific and defined by the vehicle manufacturer.	U	ASI
F182	**ApplicationDataIdentification** This value shall be used to reference the vehicle manufacturer specific ECU application data identification record. The first data byte of the record data shall be the numberOfModules that are reported. Following the numberOfModules the application data identification(s) are reported. The format of the application data identification structure shall be ECU specific and defined by the vehicle manufacturer.	U	ADI
F183	**bootSoftwareFingerprint** This value shall be used to reference the vehicle manufacturer specific ECU boot software fingerprint identification record. Record data content and format shall be ECU specific and defined by the vehicle manufacturer.	U	BSFP

Table C.1 — recordDataIdentifier data parameter definitions

Hex	Description	Cvt	Mnemonic
F184	**applicationSoftwareFingerprint** This value shall be used to reference the vehicle manufacturer specific ECU application software fingerprint identification record. Record data content and format shall be ECU specific and defined by the vehicle manufacturer.	U	ASFP
F185	**applicationDataFingerprint** This value shall be used to reference the vehicle manufacturer specific ECU application data fingerprint identification record. Record data content and format shall be ECU specific and defined by the vehicle manufacturer.	U	ADFP
F186	**reservedByDocument-Standardized** This range of values shall be reserved by this document for future definition of standardized server/vehicleIdentification options.	M	RBD
F187	**vehicleManufacturerSparePartNumber** This value shall be used to reference the vehicle manufacturer spare part number. Record data content and format shall be server specific and defined by the vehicle manufacturer.	U	VMSPN
F188	**vehicleManufacturerECUSoftwareNumber** This value shall be used to reference the vehicle manufacturer ECU (server) software number. Record data content and format shall be server specific and defined by the vehicle manufacturer.	U	VMECUSN
F189	**vehicleManufacturerECUSoftwareVersionNumber** This value shall be used to reference the vehicle manufacturer ECU (server) software version number. Record data content and format shall be server specific and defined by the vehicle manufacturer.	U	VMECUSVN
F18A	**systemSupplierIdentifier** This value shall be used to reference the system supplier name and address information. Record data content and format shall be server specific and defined by the system supplier.	U	SSID
F18B	**ECUManufacturingData** This value shall be used to reference the ECU (server) manufacturing date. Record data content and format shall be unsigned numeric, ASCII or BCD, and shall be ordered as Year, Month, Day.	U	ECUMD
F18C	**ECUSerialNumber** This value shall be used to reference the ECU (server) serial number. Record data content and format shall be server specific.	U	ECUSN
F18D	**supportedFunctionalUnits** This value shall be used to request the functional units implemented in a server.	U	SFU
F18E- F18F	**reservedByDocument-Standardized** This range of values shall be reserved by this document for future definition of standardized server/vehicleIdentification options.	M	RBD
F190	**VIN** This value shall be used to reference the VIN number. Record data content and format shall be specified by the vehicle manufacturer.	U	VIN
F191	**vehicleManufacturerECUHardwareNumber** This value shall be used by reading services to reference the vehicle manufacturer specific ECU (server) hardware number. Record data content and format shall be server specific and defined by vehicle manufacturer.	U	VMECUHN
F192	**systemSupplierECUHardwareNumber** This value shall be used to reference the system supplier specific ECU (server) hardware number. Record data content and format shall be server specific and defined by the system supplier.	U	SSECUHWN
F193	**systemSupplierECUHardwareVersionNumber**	U	SSECUHWVN

Table C.1 — recordDataIdentifier data parameter definitions

Hex	Description	Cvt	Mnemonic
	This value shall be used to reference the system supplier specific ECU (server) hardware version number. Record data content and format shall be server specific and defined by the system supplier.		
F194	**systemSupplierECUSoftwareNumber** This value shall be used to reference the system supplier specific ECU (server) software number. Record data content and format shall be server specific and defined by the system supplier.	U	SSECUSWN
F195	**systemSupplierECUSoftwareVersionNumber** This value shall be used to reference the system supplier specific ECU (server) software version number. Record data content and format shall be server specific and defined by the system supplier.	U	SSECUSWVN
F196	**exhaustRegulationOrTypeApprovalNumber** This value shall be used to reference the exhaust regulation or type approval number (valid for those systems which require type approval). Record data content and format shall be server specific and defined by the vehicle manufacturer.	U	EROTAN
F197	**systemNameOrEngineType** This value shall be used to reference the system name or engine type. Record data content and format shall be server specific and defined by the vehicle manufacturer.	U	SNOET
F198	**repairShopCodeOrTesterSerialNumber** This value shall be used to reference the repair shop code or tester (client) serial number (e.g., to indicate the most recent service client used re-program server memory). Record data content and format shall be server specific and defined by the vehicle manufacturer.	U	RSCOTSN
F199	**programmingDate** This value shall be used to reference the date when the server was last programmed. Record data content and format shall be unsigned numeric, ASCII or BCD, and shall be ordered as Year, Month, Day.	U	PD
F19A	**calibrationRepairShopCodeOrCalibrationEquipmentSerialNumber** This value shall be used to reference the repair shop code or client serial number (e.g., to indicate the most recent service client used re-calibrate the server). Record data content and format shall be server specific and defined by the vehicle manufacturer.	U	CRSCOCESN
F19B	**calibrationDate** This value shall be used to reference the date when the server was last calibrated. Record data content and format shall be unsigned numeric, ASCII or BCD, and shall be ordered as Year, Month, Day.	U	CD
F19C	**calibrationEquipmentSoftwareNumber** This value shall be used to reference software version within the client used to calibrate the server. Record data content and format shall be server specific and defined by the vehicle manufacturer.	U	CESWN
F19D	**ECUInstallationDate** This value shall be used to reference the date when the ECU (server) was installed in the vehicle. Record data content and format shall be either unsigned numeric, ASCII or BCD, and shall be ordered as Year, Month, Day.	U	EID
F19E	**ODXFileIdentifier** This value shall be used to reference the ODX (Open Diagnostic Data Exchange) file of the server to be used to interprete and scale the server data.	U	AFID
F19F	**EntityIdentifier** This value shall be used to reference the entity identifier as defined in ISO 15764 for a secured data transmission.	M	EID

Table C.1 — recordDataIdentifier data parameter definitions

Hex	Description	Cvt	Mnemonic
F1A0 - F1EF	**identificationOption-VehicleManufacturerSpecific** This range of values shall be used for vehicle manufacturer specific server/vehicle identification options.	U	IO-VMS
F1F0 - F1FF	**identificationOption-SystemSupplierSpecific** This range of values shall be used for system supplier specific server/vehicle system identification options.	U	IO-SSS
F200 – F2FF	**periodicRecordDataIdentifier** This range of values shall be used to reference periodic record data identifiers. Those can either be statically or dynamically defined.	U	PRDI
F300 – F3FF	**dynamicallyDefinedDataIdentifier** This range of values shall be used for dynamicallyDefinedDataIdentifiers.	U	DDDDI
F400 – F5FF	**OBDPids** This range of values is reserved to represent OBD/EOBD PIDs.	U	OBDPID
F600 – F7FF	**OBDMonitorIds** This range of values is reserved to represent OBD/EOBD on-board monitoring result values.	U	OBDMR
F800 – F8FF	**OBDInfoTypes** This range of values is reserved to represent OBD/EOBD info type values.	U	OBDINFTP
F900 – F9FF	**SafetySystemPIds** This range of values is reserved to represent safety related PIDs.	U	SSPID
FA00 - FCFF	**reservedForLegislativeUse** This range of values is reserved for future legislative requirements.	U	RFLU
FD00 - FEFF	**systemSupplierSpecific** This range of values shall be used to reference system supplier specific record data identifiers and input/output identifiers within the server.	U	SSS
FF00	**eraseMemory** This value shall be used to start the servers memory erase routine. The Control option and status record format shall be ECU specific and defined by the vehicle manufacturer.	U	EM
FF01	**checkProgrammingDependencies** This value shall be used to check the server's memory programming dependencies. The Control option and status record format shall be ECU specific and defined by the vehicle manufacturer.	U	CPD
FF02 - FFFF	**reservedByDocument** This range of values shall be reserved by this document for future definition.	M	RBD

C.2 scalingByte parameter definitions

The parameter scalingByte (SBYT) consists of one byte (high and low nibble). The scalingByte high nibble defines the data type, which is used to represent the recordDataIdentifier (RDI). The scalingByte low nibble defines the number of bytes used to represent the parameter in a datastream.

Table C.2 — scalingByte (High Nibble) parameter definitions

Encoding of High Nibble (Hex)	Description of Data Type	Cvt	Mnemonic

Table C.2 — scalingByte (High Nibble) parameter definitions

Encoding of High Nibble (Hex)	Description of Data Type	Cvt	Mnemonic
0	**unSignedNumeric (1 to 4 bytes)** This encoding uses a common binary weighting scheme to represent a value by mean of discrete incremental steps. One byte affords 256 steps; two bytes yields 65536 steps, etc.	U	USN
1	**signedNumeric (1 to 4 bytes)** This encoding uses a two's complement binary weighting scheme to represent a value by mean of discrete incremental steps. One byte affords 256 steps; two bytes yields 65536 steps, etc.	U	SN
2	**bitMappedReportedWithOutMask** Bit mapped encoding uses individual bits or small groups of bits to represent status. For every bit which represents status, a corresponding mask bit is required as part of the parameter definition. The mask indicates the validity of the bit for particular applications. This type of bit mapped parameter does not contain additional bytes to report the validity mask.	U	BMRWOM
3	**bitMappedReportedWithMask** Bit mapped encoding uses individual bits or small groups of bits to represent status. For every bit which represents status, a corresponding mask bit is required as part of the parameter definition. The mask indicates the validity of the bit for particular applications. This type of bit mapped parameter contains one validity mask byte for each status byte representing data.	U	BMRWM
4	**BinaryCodedDecimal** Conventional Binary Coded Decimal encoding is used to represent two numeric digits per byte. The upper nibble is used to represent the most significant digit (0 - 9), and the lower nibble the least significant digit (0 -9).	U	BCD
5	**stateEncodedVariable (1 byte)** This encoding uses a binary weighting scheme to represent up to 256 distinct states. An example is a parameter, which represents the status of the Ignition Switch. Codes "00", "01", "02" and "03" may indicate ignition off, locked, run, and start, respectively. The representation is always limited to one (1) byte.	U	SEV
6	**ASCII (1 to 15 bytes for each scalingByte)** Conventional ASCII encoding is used to represent up to 128 standard characters with the MSB = logic 0. An additional 128 custom characters may be represented with the MSB = logic 1.	U	ASCII
7	**signedFloatingPoint** Floating point encoding is used for data that needs to be represented in floating point or scientific notation. Standard IEEE formats shall be used according to ANSI/IEEE Std 754 - 1985.	U	SFP
8	**packet** Packets contain multiple data values, usually related, each with unique scaling. Scaling information is not included for the individual values.	U	P
9	**Formula** A formula is used to calculate a value from the raw data. Formula Identifiers are specified in the table definig the formulaIdentifier encoding.	U	F
A	**unit/format** The units and formats are used to present the data in a more user-friendly format. Unit and Format Identifiers are specified in the Table defining the formulaIdentifier encoding. Note: If combined units and/or formats are used, e.g. mV, then one scalingByte (and scalingData) for each unit/format shall be included in the readScalingDataByIdentifier postive response.	U	U
B - F	**reservedByDocument** Reserved by this document for future definition.	M	RBD

Table C.3 — scalingByte (Low Nibble) parameter definitions

Encoding of Low Nibble (Hex)	Description of Low Nibble	Cvt	Mnemonic
0 - F	**numberOfBytesOfParameter** This range of values specifies the number of data bytes in a data stream referenced by a parameter identifier. The length of a parameter is defined by the scaling byte(s), which is always preceded by a parameter identifier (one or multiple bytes). If multiple scaling bytes follow a parameter identifier the length of the data referenced by the parameter identifier is the summation of the content of the low nibbles in the scaling bytes. e.g. VIN is identified by a single byte parameter identifier and followed by two scaling bytes. The length is calculated up to 17 data bytes. The content of the two low nibbles may have any combination of values that add up to 17 data bytes. Note: For the scalingByte with high nibble encoded as formula or unit/format this value is $0.	U	NROBOP

C.3 scalingByteExtention parameter definitions

The parameter scalingByteExtention (SBYE) is only supported for scalingByte parameters with the high nibble encoded as formula, unit/format, or bitMappedReportedWithOutMask.

C.3.1 scalingByteExtention for scalingByte high nibble of formula

A scalingByte with high nibble encoded as formula shall be followed by scalingByteExtention bytes defining the formula. The scalingByteExtention consists a of one byte formulaIdentifier and constants as described in Table below.

Table C.4 — scalingByteExtention Bytes for formula

ScalingByteExtention Byte	Description	Cvt
#1	formulaIdentifier (refer to Table defining the formulaIdentifier encoding for details)	M
#2	C0 high byte	M
#3	C0 low byte	M
#4	C1 high byte	U
#5	C1 low byte	U
:	:	U
#2n+2	Cn high byte	U
#2n+3	Cn low byte	U

Table C.5 — formulaIdentifier Encoding

FormulaIdentifier (Hex)	Description	Cvt
00	y = C0 * x + C1	U
01	y = C0 * (x + C1)	U
02	y = C0 / (x + C1) + C2	U
03	y = x / C0 + C1	U
04	y = (x + C0) / C1	U
05	y = (x + C0) / C1 + C2	U
06	y = C0 * x	U
07	y = x / C0	U
08	y = x + C0	U
09	y = x * C0 / C1	U
0A - 7F	Reserved by document	M
80 – FF	Vehicle manufacturer specific	U

Formulas are defined using variables (y,x, etc.) and constants (C0, C1, C2, etc.). The variable y is the calculated value. The other variables, in consecutive order, are part of the data stream referenced by a recordDataIdentifier. Each constant is expressed as a two byte real number defined in the table below. The two byte real numbers (C = M * 10E) contain a 12 bit signed (2's complement) mantissa (M) and a 4 bit signed (2's complement) exponent (E). The mantissa can hold values within the range –2048 to +2047, and the exponent can scale the number by 10^{-8} to 10^{7}. The exponent is encoded in the high nibble of the high byte of the two byte real number. The mantissa is encoded in the low nibble of the low byte of the two byte real number.

Table C.6 — Two byte real number format

High Byte								Low Byte							
High Nibble				Low Nibble				High Nibble				Low Nibble			
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Exponent				Mantissa											

C.3.2 scalingByteExtention for scalingByte high nibble of unit / format

A scalingByte with high nibble encoded as unit / format shall be followed by a single scalingByteExtention byte defining the unit / format. The one byte scalingByteExtention is defined in the table below. If combined units and/or formats are used, e.g. mV, then one scalingByte (and scalingByteExtension) shall be included for each unit / format.

Table C.7 — Unit / format scalingByteExtention Encoding

ScalingByteExtention Byte #1 (Hex)	Name	Symbol	Description	Cvt
00	No unit, no prefix			U
01	Meter	m	Length	U
02	Foot	ft	Length	U
03	Inch	in	Length	U
04	Yard	yd	Length	U
05	mile (English)	mi	length	U
06	Gram	g	mass	U

Table C.7 — Unit / format scalingByteExtention Encoding

ScalingByteExtention Byte #1 (Hex)	Name	Symbol	Description	Cvt
07	ton (metric)	t	mass	U
08	Second	s	time	U
09	Minute	min	time	U
0A	Hour	h	time	U
0B	Day	d	time	U
0C	year	y	time	U
0D	ampere	A	current	U
0E	volt	V	voltage	U
0F	coulomb	C	electric charge	U
10	ohm	W	resistance	U
11	farad	F	capacitance	U
12	henry	H	inductance	U
13	siemens	S	electric conductance	U
14	weber	Wb	magnetic flux	U
15	tesla	T	magnetic flux density	U
16	kelvin	K	thermodynamic temperature	U
17	Celsius	°C	thermodynamic temperature	U
18	Fahrenheit	F	thermodynamic temperature	U
19	candela	cd	luminous intensity	U
1A	radian	rad	plane angle	U
1B	degree	°	plane angle	U
1C	hertz	Hz	frequency	U
1D	joule	J	energy	U
1E	Newton	N	force	U
1F	kilopond	kp	force	U
20	pound force	lbf	force	U
21	watt	W	power	U
22	horse power (metric)	hk	power	U
23	horse power (UK and US)	hp	power	U
24	Pascal	Pa	pressure	U
25	bar	bar	pressure	U
26	atmosphere	atm	pressure	U
27	pound force per square inch	psi	pressure	U
28	becqerel	Bq	radioactivity	U
29	lumen	lm	light flux	U
2A	lux	lx	illuminance	U
2B	liter	l	volume	U
2C	gallon (British)	-	volume	U
2D	gallon (US liq)	-	volume	U
2E	cubic inch	cu in	volume	U
2F	meter per second	m/s	speed	U
30	kilometer per hour	km/h	speed	U
31	mile per hour	mph	speed	U
32	revolutions per second	rps	angular velocity	U

Table C.7 — Unit / format scalingByteExtention Encoding

ScalingByteExtention Byte #1 (Hex)	Name	Symbol	Description	Cvt
33	revolutions per minute	rpm	angular velocity	U
34	counts	-	-	U
35	percent	%	-	U
36	milligram per stroke	mg/stroke	mass per engine stroke	U
37	meter per square second	m/s^2	acceleration	U
38	Newton meter	Nm	moment (e.g. torsion moment)	U
39	liter per minute	l/min	flow	U
3A	Watt per square meter	W/m^2	Intensity	U
3B	Bar per second	bar/s	Pressure change	U
3C	Radians per second	rad/s	Angular velocity	U
3D	Radians per square second	rad/s^2	Angular acceleration	U
3E	Kilogram per square meter	kg/m^2	-	U
3F	-	-	Reserved by document	M
40	exa (prefix)	E	10^{18}	U
41	peta (prefix)	P	10^{15}	U
42	tera (prefix)	T	10^{12}	U
43	giga (prefix)	G	10^{9}	U
44	mega (prefix)	M	10^{6}	U
45	kilo (prefix)	k	10^{3}	U
46	hecto (prefix)	h	10^{2}	U
47	deca (prefix)	da	10	U
48	deci (prefix)	d	10^{-1}	U
49	centi (prefix)	c	10^{-2}	U
4A	milli (prefix)	m	10^{-3}	U
4B	micro (prefix)	m	10^{-6}	U
4C	nano (prefix)	n	10^{-9}	U
4D	pico (prefix)	p	10^{-12}	U
4E	femto (prefix)	f	10^{-15}	U
4F	atto (prefix)	a	10^{-18}	U
50	Date1	-	Year-Month-Day	U
51	Date2	-	Day/Month/Year	U
52	Date3	-	Month/Day/Year	U
53	week	W	calendar week	U
54	Time1	-	UTC Hour/Minute/Second	U
55	Time2	-	Hour/Minute/Second	U
56	DateAndTime1	-	Second/Minute/Hour/Day/Month/Y	
57	DateAndTime2	-	Second/Minute/Hour/Day/Month/Y	U
58-FF	-	-	Reserved by document	M

C.3.3 scalingByteExtention for scalingByte high nibble of bitMappedReportedWithOutMask

A scalingByte with high nibble encoded as bitMappedReportedWitOutMask shall be followed by scalingByteExtention bytes representing the validity mask for the bit mapped recordDataIdentifier. Each byte shall indicate which bits of the corresponding recordDataIdentifier byte are supported for the current application.

Table C.8 — scalingByteExtention for bitMappedReportedWithOutMask

scalingByteExtention Byte	Description	Cvt
#1	recordDataIdentifier dataRecord#1 validity mask	M
:	:	C_1
#p	recordDataIdentifier dataRecord#p validity mask	C_1
C_1: The presence of this parameter depends on the size of the recordDataIdentifier the information is being requested for. The validity mask shall have as many bytes as the recordDataIdentifier has dataRecords.		

C.4 transmissionMode parameter definitions

Table C.9 — transmissionMode parameter definitions

Hex	Description	Cvt	Mnemonic
00	**reservedByDocument** This value shall be reserved by this document for future definition.	M	RBD
01	**sendAtSlowRate** This parameter specifies that the server shall transmit the requested dataRecord information at a slow rate in response to the request message (where the # responses to be sent = maximumNumberOfResponsesToSend). The repetition rate specified by the transmissionMode parameter slow is vehicle manufacturer specific, and pre-defined in the server.	U	SASR
02	**sendAtMediumRate** This parameter specifies that the server shall transmit the requested dataRecord information at a medium rate in response to the request message (where the # responses to be sent = maximumNumberOfResponsesToSend). The repetition rate specified by the transmissionMode parameter medium is vehicle manufacturer specific, and pre-defined in the server.	U	SAMR
03	**sendAtFastRate** This parameter specifies that the server shall transmit the requested dataRecord information at a fast rate in response to the request message (where the # responses to be sent = maximumNumberOfResponsesToSend). The repetition rate specified by the transmissionMode parameter fast is vehicle manufacturer specific, and pre-defined in the server.	U	SAFR
04	**stopSending** The server stops transmitting positive response messages send periodically/repeatedly. Note that maximumNumberOfResponsesToSend parameter should be set to 01 hex if transmissionMode = stopSending (otherwise, server operation could be undefined).	C	SS
05 – FF	**reservedByDocument** This value shall be reserved by this document for future definition.	M	RBD
C stopSending shall be supported if sendAtSlowRate, sendAtMediumRate, and/or sendAtFastRate are supported.			

Annex D
(normative)

Stored data transmission functional unit data parameter definitions

This annex includes all data parameters of the services specified in the “Stored Data Transmission Functional Unit”. Included are definitions for groupOfDTC, DTCFailureTypeByte, DTCStatusMask / statusOfDTC, and a timing diagram to illustrate the behavior of DTC status bits.

D.1 groupOfDTC parameter definition

The following table provides group of DTC definitions.

Table D.1 — Definition of groupOfDTC and range of DTC numbers

Hex	Description	Cvt	Mnemonic
000000	Emissions-related systems	C	ERS
to be determined by vehicle manufacturer	Powertrain Group: engine and transmission	U	PG
	Powertrain DTCs	U	PDTC_
	Chassis Group	U	CG
	Chassis DTCs	U	CDTC_
	Body Group	U	BG
	Body DTCs	U	BDTC_
	Network Communication Group	U	NCG
	Network Communication DTCs	U	NCDTC_
FFFFFF	All Groups (all DTCs)	M	AG
C = Conditional: this parameter selects the emissions-related systems.			

D.2 DTCFailureTypeByte parameter definition

The DTC Failure Type consists of sixteen (16) different Failure Categories, where each category is associated with sixteen (16) Sub Type Failures (also known as symptoms). The Sub Type Failures are logically grouped in a DTC Failure Type Category. This shall simplify the selection of the appropriate Sub Type Failure {Symptom} for a DTC.

The DTC Failure Category is coded in the High Nibble of the “DTCFailureTypeByte” and the Failure Sub Type is coded in the Low Nibble of the “DTCFailureTypeByte”.

Table D.2 — DTCFailureTypeByte definition

DTCFailureTypeByte definition	
High Nibble (bit 7-4, 0x - Fx hex)	**Low Nibble (bit 3-0, x0 - xF hex)**
DTC Failure Category	DTC Failure Sub Type

Existing ISO documents for defining DTCs such as ISO 15031-6 currently define DTCs for existing systems. If a standard DTC is already defined for a component / system and that DTC description already comprehends the DTC Failure Type information, then the standard DTC number can be used and the DTCFailureTypeByte shall be set to a value of 00 hex. A DTCFailureTypeByte value of 00 hex indicates that no additional information is contained in the DTCFailureTypeByte.

The following example shows the three (3) principle combinations of DTC and DTCFailureTypeByte.

— DTC which does not require any additional description included in the DTCFailureTypeByte (no DTC Failure Category name and no DTC Failure Sub Type) e.g. emissions-related DTC (012700 hex): P0127 Intake Air Temperature Too High,

— DTC which requires additional description included in the DTCFailureTypeByte (DTC Failure Category name and no DTC Failure Sub Type) e.g., DTC (011501): P0115 Engine Coolant Temperature Circuit – General Electrical Failure,

— DTC, which requires additional description included in the DTCFailureTypeByte (DTC Failure Category name and DTC Failure Sub Type) e.g., DTC (010023): P0100 Mass or Volume Air Flow Circuit – General Signal Failure - signal stuck low.

D.2.1 DTC Failure Category Definition

The table below specifies the “DTC Failure Categories”.

Table D.3 — DTC Failure Category definition

DTCFailureTypeByte Category Definitions		
High Nibble (0000b - 1111b)	**Category # (hex)**	**Category Description**
0000	0	**General Failure Information** This category includes all other categories and is used when the fault within that failure category is unique (not amenable to standardization through assignment of a new Sub Type) or when the detected fault is best described by two or more Sub Types within that Failure Category.
0001	1	**General Electrical Failures** This category includes standard wiring failure modes (i.e., shorts and opens), and direct current (DC) quantities related by Ohm's Law.
0010	2	**General Signal Failures** This category includes quantities related to amplitude, frequency or rate of change, and waveshape.
0011	3	**FM (Frequency Modulated) / PWM (Pulse Width Modulated) Failures** This category includes faults related to Frequency Modulated (FM) and Pulse Width Modulated (PWM) inputs and outputs of the server. This category also includes faults where position is determined by counts.
0100	4	**System Internal Failures** This category includes faults related to memory, software, and internal electrical circuitry; requiring component (server, sensor, etc.) replacement.
0101	5	**System Programming Failures** This category includes faults related to operational software, calibrations, and options; remedied by configuring/programming a part of the system (server, sensor, etc.).
0110	6	**Algorithm Based Failures** This category includes faults based on comparing two or more input parameters for plausibility or comparing a single parameter to itself with respect to time.
0111	7	**Mechanical Failures** This category includes faults detected by inappropriate motion in response to server related

Table D.3 — DTC Failure Category definition

DTCFailureTypeByte Category Definitions		
High Nibble (0000b - 1111b)	**Category # (hex)**	**Category Description**
		input/controlled output.
1000	8	**Bus Signal / Message Failures** This category includes faults related to bus hardware and signal integrity. This category is also used when the physical input for a signal is located in one server and another server diagnoses the circuit or inhibits operation due to a reported failure of that circuit.
1001	9	**Component Failures** This category includes faults related to component failures (including parametric, performance assembly and operating enviroment failures).
1010 – 1110	A - E	**Reserved by document** This range of values is reserved by the document for future expansion.
1111	F	**Vehicle manufacturer / System supplier specific** This category is reserved for vehicle manufacturer/system supplier use.

The tables below specify the different “DTC Failure Sub Types”. All failure sub types shall be assigned as mutually exclusive, i.e., a single point failure shall result in a single base DTC and failure type byte combination. When a detected fault can logically be assigned to two or more Failure Categories, use the lowest number category that is applicable to the fault. The exception to this rule is the condition where the lowest number failure category requires a failure category of 0 (hex). In this case, use the lowest number failure category with a non-zero value. Similarly, within the Failure Category, use the lowest number Sub Type applicable to the fault. The Failure Category, "General Failure Information", should be used when the fault within that failure category is unique (not amenable to standardization through assignment of a new Sub Type) or when the detected fault is best described by two or more Sub Types within that Failure Category.

D.2.2 DTC Failure Sub Type definition of General Failure Information

This category includes all other categories and is used when the fault within that failure category is unique (not amenable to standardization through assignment of a new Sub Type) or when the detected fault is best described by two or more Sub Types within that Failure Category.

Table D.4 — DTC Failure Sub Type definition for failure category ‘0’

Failure Type Byte (hex)	Sub Type Nibble (binary)	General Failure Information Sub Type Description
00	0000	**no sub type information** This sub type is used for failures where the base DTC text string provides the complete description of the failure itself (no Category and no Sub Type information used, e.g. emissions-related DTC (012700 hex): P0127 Intake Air Temperature Too High).
01	0001	**General Electrical Failure** This sub type is used for General Electrical Failures that cannot be assigned to a specific sub type (Category information and no Sub Type information, e.g. DTC (011501): P0115 Engine Coolant Temperature Circuit – General Electrical Failure).
02	0010	**General signal failure** This sub type is used for General Signal Failures that cannot be assigned to a specific sub type (Category information and no Sub Type information, e.g. DTC (014802): P0148 Fuel Delivery Error – General Signal Failure).
03	0011	**FM (Frequency Modulated) / PWM (Pulse Width Modulated) Failures** This sub type is used for FM / PWM Failures that cannot be assigned to a specific sub

Table D.4 — DTC Failure Sub Type definition for failure category '0'

Failure Type Byte (hex)	Sub Type Nibble (binary)	General Failure Information Sub Type Description
		type.
04	0100	**System Internal Failures** This sub type is used for server Internal Failures that cannot be assigned to a specific sub type.
05	0101	**System Programming Failures** This sub type is used for System Programming Failures that cannot be assigned to a specific sub type.
06	0110	**Algorithm Based Failures** This sub type is used for Algorithm Based Failures that cannot be assigned to a specific sub type.
07	0111	**Mechanical Failures** This sub type is used for Mechanical Failures that cannot be assigned to a specific sub type.
08	1000	**Bus Signal / Message Failures** This sub type is used for Bus Signal / Message Failures that cannot be assigned to a specific sub type.
09	1001	**Component Failures** This sub type is used for Component Failures that cannot be assigned to a specific sub type.
0A – 0F	1010 – 1111	**reserved by document** This range of values is reserved by the document for future expansion.

D.2.3 DTC Failure Sub Type definition of general electrical failures

The table below specifies the standard wiring failure modes (i.e., shorts and opens), and direct current (DC) quantities related by Ohm's Law.

Table D.5 — DTC Failure Sub Type definition for failure category '1'

Failure Type Byte (hex)	Sub Type Nibble (binary)	General Electrical Failures Sub Type Description
10	0000	**reserved by document** This value is reserved by the document for future expansion.
11	0001	**circuit short to ground** This sub type is used for failures, where the server measures ground (battery negative) potential for greater than a specified time period or when some other value is expected.
12	0010	**circuit short to battery** This sub type is used for failures, where the server measures vehicle system (battery positive) potential for greater than a specified time period or when some other value is expected.

Table D.5 — DTC Failure Sub Type definition for failure category '1'

Failure Type Byte (hex)	Sub Type Nibble (binary)	General Electrical Failures Sub Type Description
13	0011	**circuit open** This sub type is used for failures, where the server determines an open circuit via lack of bias voltage, low current flow, no change in the state of an input in response to an output, etc.
14	0100	**circuit short to ground or open** This sub type is used for failures, where the condition detected by the server is the same for either indicated failure mode.
15	0101	**circuit short to battery or open** This sub type is used for failures, where the condition detected by the server is the same for either indicated failure mode.
16	0110	**circuit voltage below threshold** This sub type is used for failures, where the server measures a voltage below a specified range but not necessarily a short to ground.
17	0111	**circuit voltage above threshold** This sub type is used for failures where, the server measures a voltage above a specified range but not necessarily a short to battery.
18	1000	**circuit current below threshold** This sub type is used for failures, where the server measures current flow below a specified range.
19	1001	**circuit current above threshold** This sub type is used for failures, where the server measures current flow above a specified range.
1A	1010	**circuit resistance below threshold** This sub type is used for failures, where the server infers a circuit resistance below a specified range.
1B	1011	**circuit resistance above threshold** This sub type is used for failures, where the server infers a circuit resistance above a specified range.
1C	1100	**circuit voltage out of range** This sub type is used for failures, where the server measures a voltage outside the expected range but not identified as too high or too low.
1D	1101	**circuit current out of range** This sub type is used for failures, where the server measures a current outside the expected range but not identified as too high or too low.
1E	1110	**circuit resistance out of range** This sub type is used for failures, where the server measures a resistance outside the expected range but not identified as too high or too low.
1F	1111	**circuit intermittent** This sub type is used for failures, where the server momentarily detects one of the conditions defined above, but not long enough to set a specific sub type.

D.2.4 DTC Failure Sub Type definition of general signal failures

The table below specifies quantities related to amplitude, frequency or rate of change, and waveshape.

Table D.6 — DTC Failure Sub Type definition for failure category '2'

Failure Type Byte (hex)	Sub Type Nibble (binary)	General Signal Failures Sub Type Description
20	0000	**reserved by document** This value is reserved by the document for future expansion.
21	0001	**signal amplitude < minimum** This sub type is used for failures where the server measures a signal voltage below a specified range but not necessarily a short to ground (e.g., low gain).
22	0010	**signal amplitude > maximum** This sub type is used for failures where the server measures a signal voltage above a specified range but not necessarily a short to battery (e.g., gain too high).
23	0011	**signal stuck low** This sub type is used for failures where the server measures a signal that remains low when transitions are expected.
24	0100	**signal stuck high** This sub type is used for failures where the server measures a signal that remains high when transitions are expected.
25	0101	**signal shape / waveform failure** This sub type is used for failures where the shape of the signal (plot of the amplitude with respect to time) is not correct, e.g., improper circuit impedance.
26	0110	**signal rate of change below threshold** This sub type is used for failures where the signal transitions more slowly than is reasonably allowed.
27	0111	**signal rate of change above threshold** This sub type is used for failures where the signal transitions more quickly than is reasonably allowed.
28	1000	**signal bias level out of range / zero adjustment failure** This sub type is used for failures where the server applies a bias voltage to a circuit upon which is superimposed a signal voltage (e.g., Oxygen Sensor circuit.). This sub type is also used for failures where the server applies a zero signal level to a circuit upon which is superimposed a signal voltage (e.g., bias voltage to an Oxygen Sensor circuit, or a filtered digital m/sec^2 signal while vehicle stands still for a lateral accelerator sensor module.)
29	1010	**signal signal invalid** This sub type is used for failures where the value of the signal is not plausible given the operating conditions.
2A - 2E	1001	**reserved by document** This range of values is reserved by the document for future expansion.
2F	1111	**signal erratic** This sub type is used for failures where the signal is momentarily implausible (not long enough for "signal invalid") or discontinuous.

D.2.5 DTC Failure Sub Type definition of FM (Frequency Modulation) / PWM (Pulse Width Modulation) failures

The table below specifies faults related to Frequency Modulated (FM) and Pulse Width Modulated (PWM) inputs and outputs of the server. This category also includes faults where position is determined by counts.

Table D.7 — DTC Failure Sub Type definition for failure category '3'

Failure Type Byte (hex)	Sub Type Nible (binary)	FM (Frequency Modulated) / PWM (Pulse Width Modulated) Failures Sub Type Description
30	0000	**reserved by document** This value is reserved by the document for future expansion.
31	0001	**no signal** This sub type is used for failures where the server does not detect a signal which ought to be present (e.g., wheel speed signals present for three of the four wheels and brakes not applied.)
32	0010	**signal low time < minimum** This sub type is used for failures where the server detects the low pulse is too narrow with respect to time.
33	0011	**signal low time > maximum** This sub type is used for failures where the server detects the low pulse is too wide with respect to time.
34	0100	**signal high time < minimum** This sub type is used for failures where the server detects the high pulse is too narrow with respect to time.
35	0101	**signal high time > maximum** This sub type is used for failures where the server detects the high pulse is too wide with respect to time.
36	0110	**signal frequency too low** This sub type is used for failures where the server detects excessive duration for one cycle of the output across a specified sample size.
37	0111	**signal frequency too high** This sub type is used for failures where the server detects insufficient duration for one cycle of the output across a specified sample size.
38	1000	**signal frequency incorrect** This sub type is used for failures where the server measures an incorrect number of cycles in a given time period.
39	1001	**incorrect has too few pulses** This sub type is used for failures where the server measures too few pulses (e.g., position is calibrated in counts from one extreme to the other).
3A	1010	**incorrect has too many pulses** This sub type is used for failures where the server measures too many pulses (e.g., position is calibrated in counts from one extreme to the other).
3B - 3F	1011 - FFF	**reserved by document** This range of values is reserved by the document for future expansion.

D.2.6 DTC Failure Sub Type definition of system internal failures

The table below specifies faults related to memory, software, and internal electrical circuitry; requiring component (server, sensor, etc.) replacement.

Table D.8 — DTC Failure Sub Type definition for failure category '4'

Failure Type Byte (hex)	Sub Type Nibble (binary)	System Internal Failures Sub Type Description
40	0000	**reserved by document** This value is reserved by the document for future expansion.
41	0001	**general checksum failure** This sub type is used by the server to indicate an incorrect checksum calculation where memory type is not specified.
42	0010	**general memory failure** This sub type is used by the server to indicate a memory failure where memory type is not specified.
43	0011	**special memory failure** This sub type is used by the server to indicate a memory failure where the specific memory type is not defined in this category.
44	0100	**data memory failure** This sub type is used by the server to indicate a data (or working) memory failure for embedded systems using FLASH memory. This is equivalent to RAM in RAM/ROM/EEPROM embedded systems.
45	0101	**program memory failure** This sub type is used by the server to indicate a progam memory failure for embedded systems using FLASH memory. This is equivalent to ROM in RAM/ROM/EEPROM embedded systems.
46	0110	**calibration / parameter memory failure** This sub type is used by the server to indicate a calibration / parameter memory failure for embedded systems using FLASH memory. This is equivalent to EEPROM in RAM/ROM/EEPROM embedded systems.
47	0111	**watchdog / safety µC failure** This sub type is used by the server to indicate a watchdog / safety µC failure.
48	1000	**supervision software failure** This sub type is used by the server to indicate a supervision software failure.
49	1001	**internal electronic failure** This sub type is used by the server to indicate the detection of an internal circuit failure.
4A	1010	**incorrect component installed** This sub type is used by the server to indicate a mismatch between the hardware connected to the server and the hardware expected by the server.
4B	1011	**over temperature** This sub type is used by the server to indicate the detection of an internal temperature above the expected range.
4C - 4F	1101 - 1111	**reserved by document** This range of values is reserved by the document for future expansion.

D.2.7 DTC Failure Sub Type definition of system programming failures

The table below specifies faults related to operational software, calibrations, and options; remedied by configuring/programming a part of the system (server, sensor, etc.).

Table D.9 — DTC Failure Sub Type definition for failure category '5'

Failure Type Byte (hex)	Sub Type Nibble (binary)	System Programming Failures Sub Type Description
50	0000	**reserved by document**
		This value is reserved by the document for future expansion.
51	0001	**not programmed**
		This sub type is used by the server to indicate that programming is required.
52	0010	**not activated**
		This sub type is used by the server to indicate that that some portion of the program has not been enabled.
53	0011	**deactivated**
		This sub type is used by the server to indicate that that some portion of the program has been disabled.
54	0100	**missing calibration**
		This sub type is used by the server to indicate, that an operational range etc. for a sensor or actuator must be taught to the server, e.g. by programming or learning.
55	0101	**not configured**
		This sub type is used by the server to indicate the need to enter (program) the sub system option content or the vehicle option content.
56 - 5F	0101 - 1111	**reserved by document**
		This range of values is reserved by the document for future expansion.

D.2.8 DTC Failure Sub Type definition of algorithm based failures

The table below specifies faults based on comparing two or more input parameters for plausibility, comparing a single parameter to itself with respect to time, or inhibits operation due to a reported failure of that circuit.

Table D.10 — DTC Failure Sub Type definition for failure category '6'

Failure Type Byte (hex)	Sub Type Nibble (binary)	Algorithm Based Failures Sub Type Description
60	0000	**reserved by document** This value is reserved by the document for future expansion.
61	0001	**signal calculation failure** This sub type is used for algorithm based calculation failures.
62	0010	**signal compare failure** This sub type is used for failures where the server compares two or more input parameters for plausibility.
63	0011	**circuit / component protection time-out** This sub type is used for failures where the server detects a function is active for greater than a specified time period.
64	0100	**signal plausibility failure** This sub type is used for failures where the server detects plausibility failures.
65	0101	**signal has too few transitions / events** This sub type is used for failures where the server monitors a parameter over time within specified limits and detects fewer than the expected number of transitions.

Table D.10 — DTC Failure Sub Type definition for failure category '6'

Failure Type Byte (hex)	Sub Type Nibble (binary)	Algorithm Based Failures Sub Type Description
66	0110	**signal has too many transitions / events** This sub type is used for failures where the server monitors a parameter over time within specified limits and detects more than the expected number of transitions.
67	0111	**signal incorrect after event** This sub type is used for failures where the server does not see the correct change of a parameter or group of parameters in response to a particular event.
68	1000	**event information** This sub type is used by the server to indicate the detection of a system event that was not caused by the server itself but forces the server to store a DTC (e.g. missing functionality from another system/server).
69 - 6F	1001 - 1111	**reserved by document** This range of values is reserved by the document for future expansion.

D.2.9 DTC Failure Sub Type definition of mechanical failures

The table below specifies faults detected by inappropriate motion in response to server related input/controlled output.

Table D.11 — DTC Failure Sub Type definition for failure category '7'

Failure Type Byte (hex)	Sub Type Nibble (binary)	Mechanical Failures Sub Type Description
70	0000	**reserved by document** This value is reserved by the document for future expansion.
71	0001	**actuator stuck** This sub type is used for failures where the server does not detect any motion in response to energizing a motor, solenoid, relay, etc.
72	0010	**actuator stuck open** This sub type is used for failures where the server does not detect any motion upon commanding the operation of a motor, solenoid, relay, etc., to close some piece of equipment.
73	0011	**actuator stuck closed** This sub type is used for failures where the server does not detect any motion upon commanding the operation of a motor, solenoid, relay, etc., to open some piece of equipment.
74	0100	**actuator slipping** This sub type is used for failures where the server detects excessive duration to command a motor, solenoid, relay, etc., to move a piece of equipment to a desired position.
75	0101	**emergency position not reachable** This sub type is used for failures where the server is unable to command a motor, solenoid, relay, etc., to move a piece of equipment to the emergency position.
76	0110	**wrong mounting position** This sub type is used for failures where the server detects incorrectly mounted components, e.g., acceleration sensor showing a position error of 90°.
77	0111	**commanded position not reachable**

Table D.11 — DTC Failure Sub Type definition for failure category '7'

Failure Type Byte (hex)	Sub Type Nibble (binary)	Mechanical Failures Sub Type Description
		This sub type is used for failures where the server is unable to command a motor, solenoid, relay, etc., to move a piece of equipment to the commanded position either due to a failure in the actuator or its mechanical environment.
78	1000	**alignment or adjustment incorrect** This sub type is used for failures where the server detects incorrectly adjusted or aligned components
79	1001	**mechanical linkage failure** This sub type is used for failures where the server detects that the actuator is operational but the driven device is not operating e.g. drive cable for power sliding door broken
7A	1010	**fluid leak or seal failure** This sub type is used for failures where the server detects that a mechanical component has an unexpected gas or liquid flow in, out or through the component.
7B	1011	**low fluid level** This sub type is used for failures where the server detects that a fluid level is too low for proper operation of the system
7C - 7F	1100 - 1111	**reserved by document** This range of values is reserved by the document for future expansion.

D.2.10 DTC Failure Sub Type definition of bus signal failures

The table below specifies faults related to bus hardware and signal integrity. This category is also used when the physical input for a signal is located in one server and another server diagnoses the circuit.

Table D.12 — DTC Failure Sub Type definition for failure category '8'

Failure Type Byte (hex)	Sub Type Nibble (binary)	Bus Signal / Message Failures Sub Type Description
80	0000	**reserved by document**
		This value is reserved by the document for future expansion.
81	0001	**invalid serial data received**
		This sub type is used by the server to indicate a signal was received with the corresponding validity bit equal to "invalid" or post processing of the signal determines it is invalid.
82	0010	**alive / sequence counter incorrect / not updated**
		This sub type is used by the server to indicate, that a signal was received without the corresponding rolling count value being properly updated.

Table D.12 — DTC Failure Sub Type definition for failure category '8'

Failure Type Byte (hex)	Sub Type Nibble (binary)	Bus Signal / Message Failures Sub Type Description
83	0011	**value of signal protection calculation incorrect**
		This sub type is used by the server to indicate, that a message was processed with an incorrect protection (checksum) calculation.
84	0100	**signal below allowable range**
		This sub type is used for failures where some circuit quantity, reported via serial data, is below a specified range.
85	0101	**signal above allowable range**
		This sub type is used for failures where some circuit quantity, reported via serial data, is above a specified range.
86	0110	**signal invalid**
		This sub type is used for failures where some circuit quantity, reported via serial data, is not plausible given the operating conditions.
87	0111	**missing message**
		This sub type is used for failures where one (or more) expected message(s) is not received, e.g., periodic transmission where the repetition time is too high, or message not received as a result of unforeseen reset events of the concerning component (e.g. engine control unit communicating with ABS).
88	1000	**bus off**
		This sub type is used for failures where a data bus is not available.
89 - 8E	1001 – 1110	**reserved by document**
		This range of values is reserved by the document for future expansion.
8F	1111	**erratic**
		This sub type is used for failures where the signal, reported via serial data, is momentarily implausible or discontinuous.

D.2.11 DTC Failure Sub Type definition of component failures

The table below specifies faults related to component failures.

Table D.13 — DTC Failure Sub Type definition for failure category '9'

Failure Type Byte (hex)	Sub Type Nibble (binary)	Component Failures Sub Type Description
90	0000	**reserved by document** This value is reserved by the document for future expansion.
91	0001	**parametric** This sub type is used for failures where the server has detected that a component parameter e.g. capacitance or inductance is outside its expected range.
92	0010	**performance or incorrect operation** This sub type is used for failures where the server has detected that the component performance is outside its expected range or operating in an incorrect way.

Table D.13 — DTC Failure Sub Type definition for failure category '9'

Failure Type Byte (hex)	Sub Type Nibble (binary)	Component Failures Sub Type Description
93	0011	**no operation** This sub type is used for failures where the server has detected that the component is not operating.
94	0100	**unexpected operation** This sub type is used for failures where the server has detected that the component is operating in a way or at a time that it has not been commanded to operate.
95	0101	**incorrect assembly** This sub type is used for failures where the server has detected that the component has been incorrectly installed e.g. hydraulic pipes crossed over, circuits cross wired or polarity errors.
96	0110	**component internal failure** This sub type is used for failures where the server has recived an indication about the component that indicates a failure e.g. an intelligent actuator or sensor is indicating an internal fault.
97	0111	**Component or system operation obstructed or blocked** This sub type is used for failures where the server has detected that the operation of a component is prevented by an obstruction e.g. advanced cruise system radar beam obstructed.
98	1000	**component or system over temperature** This sub type is used for failures where the server has detected that the temperature is too high for the correct operation of the component or system.
99 – 9F	1001 - 1111	**reserved by document** This range of values is reserved by the document for future expansion.

D.3 DTCSeverityMask and DTCSeverity bit definitions

This section defines the mapping of the DTCSeverityMask / DTCSeverity parameters used with the ReadDTCInformation service. Every server shall adhere to the convention for storing bit-packed DTC severity information as defined in the table below.

The severity information is reported in a 1-byte value. Only the upper 3 bits (bit 7-5) of the 1-byte value are used to represent the DTC severity information. The remaining bits (bit 4-0) have to be set to zero (0). Based on this the following bit coding applies to the 3 bit DTC severity information contained in the 1-byte DTCSeverity parameter.

Table D.14 —DTCSeverity byte definition

DTCSeverity byte							
Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
3 bit severity information			reserved by document - to be set to zero (0)				

Table D.15 —DTC severity bit definitions (bit 7-5)

Bit 7-5	Description	Cvt	Mnemonic
000b	**noSeverityAvailable** There is no severity information available.	M	NSA
001b	**maintenanceOnly** This value indicates that the failure requests maintenance only.	M	MO
010b	**checkAtNextHalt** This value indicates to the failure requires a check of the vehicle at the nex halt.	M	CHKANH
100b	**checkImmediately** This value indicates to the failure requires an immediate check of the vehicle.	M	CHKI

D.4 DTC functional unit definitions

The DTCFunctionalUnit are implementation specific and shall be specified in the respective implementation standard.

D.5 DTCStatusMask and statusOfDTC bit definitions

This section defines the mapping of the DTCStatusMask / statusOfDTC parameters used with the ReadDTCInformation service. Every server shall adhere to the convention for storing bit-packed DTC status information as defined in table below. Actual usage of the bit-fields shall be vehicle manufacturer specific.

The following is list of definitions used for the description of the DTC status bit definitions.

— Test – A test is an on-board diagnostic software algorithm that determines the malfunction status of a component or system. Some tests (non-continuous) run only once during a monitoring cycle. Other tests (continuous tests) can run every program loop, as often as every few milliseconds. Normally, continuous tests must detect a fault for a period of time (for example, 5 seconds) before the testFailed bit is set and the component is considered to be malfunctioning. Each test is normally associated with a unique DTC (and fault symptom when applicable).

— Failure – A failure is the inability of a component or system to meet its intended function. A failure has occurred when fault conditions have been detected for a sufficient period of time to warrant storage of a pending or confirmed DTC. The terms "failure" and "malfunction" are interchangeable.

— Monitor – A monitor consists of one or more tests used to determine the malfunction status of a component or system.

— Monitoring cycle – A monitoring cycle is a manufacturer-defined set of conditions during which a test can run. For many body and chassis modules, a monitoring cycle consists of powering up and powering down the module. For powertrain modules, there are usually additional criteria to define monitoring cycles. Most powertrain modules use an engine-running or engine-off time period to define a monitoring cycle. For example, an engine-running monitoring cycle for a particular manufacturer may begin after the engine is started and continue until the server internally powers down. If a bit is set for a test that completed during the current monitoring cycle, the bit must remain set until the server powers down (except bit 0 for continuous monitors). The bit must then be cleared at the next engine start-up (beginning of the new monitoring cycle). An engine-off monitoring cycle for another manufacturer may begin after the key is turned off. If a bit is set for a test that completed during the current monitoring cycle, the bit must remain set until the key is turned off again (beginning of the new monitoring cycle). Note: These are only examples; manufacturers can define other cycles as long as they meet legislated requirements for emission-based monitors.

— Complete – Complete is an indication that a test was able to determine whether a malfunction exists or does not exist for the current monitoring cycle. (Complete does not imply failed.)

Table D.16 — DTC status bit definitions

Bit	Description	Cvt		Mnemonic
		Emission	Non-Emission	
0	**testFailed** This bit shall indicate the result of the most recently performed test. A logical '1' shall indicate that the last test failed. Reset to 0 if the result of the most recently performed test returns a "pass" result, or after a call has been made to ClearDiagnosticInformation. Additional reset conditions may be defined by the vehicle manufacturer / implementation. 1 = most recent return from DTC test indicated a failed result. 0 = most recent return from DTC test indicated no failure detected.	U	U	TF
1	**testFailedThisMonitoringCycle** This bit shall indicate whether or not a diagnostic test has reported a testFailed result at any time during the current monitoring cycle (or that a testFailed result has been reported during the current monitoring cycle and after the last time a call was made to ClearDiagnosticInformation). Reset to 0 when a new monitoring cycle is initiated or after a call to ClearDiagnosticInformation. Note that once this bit is set to 1, it shall remain a 1 until a new monitoring cycle is started. 1 = testFailed: result was reported at least once during the current monitoring cycle. 0 = testFailed: result has not been reported during the current monitoring cycle or after a call was made to ClearDiagnosticInformation during the current monitoring cycle.	M	C_1	TFTMC
2	**pendingDTC** This bit shall indicate whether or not a diagnostic test has reported a testFailed result at any time during the current or last completed monitoring cycle. The status shall only be updated only if the test runs and completes. The criteria to set the pendingDTC bit and the TestFailedThisMonitoringCycle bit are the same. The difference is that the testFailedThisMonitoringCycle is cleared at the end of the current monitoring cycle and the pendingDTC bit is not cleared until a monitoring cycle has completed where the test has passed at least once and never failed. If the test did not complete during the current monitoring cycle, the status bit shall not be changed. For example, if a monitor stops running after a confirmed DTC is set, the pendingDTC must remain set =1. For an OBD DTC, a pending DTC is required to be stored after a malfunction is detected during the first monitoring cycle. 1 = This bit shall be set to 1 and latched if a malfunction is detected during the current monitoring cycle. 0 = This bit shall be set to 0 after completing a monitoring cycle during which the test completed and a malfunction was not detected or upon a call to the ClearDiagnosticInformation service.	M	U	PDTC

Table D.16 — DTC status bit definitions

Bit	Description	Cvt		Mnemonic
		Emission	Non-Emission	
3	**confirmedDTC** This bit shall indicate whether a malfunction was detected enough times to warrant that the DTC is stored in long-term memory (pendingDTC has been set = 1 one or more times, depending on the DTC confirmation criteria). A confirmedDTC does not always indicate that the malfunction is necessarily present at the time of the request. (PendingDTC or testFailedThisMonitoringCycle can be used to determine if a malfunction is present at the time of the request.). Reset to 0 after a call to ClearDiagnosticInformation or after aging criteria has been satisfied (e.g., 40 engine warm-ups without another detected malfunction). Note: DTC confirmation and aging criteria are defined by the vehicle manufacturer or mandated by On Board Diagnostic regulations. 1 = DTC confirmed at least once since the last call to ClearDiagnosticInformation and aging criteria have not yet been satisfied. 0 = DTC has never been confirmed since the last call to ClearDiagnosticInformation or after the aging criteria have been satisfied for the DTC.	M	M	CDTC
4	**testNotCompletedSinceLastClear** This bit shall indicate whether a DTC test has ever run and completed since the last time a call was made to ClearDiagnosticInformation. 1 shall indicate that the DTC test has not run to completion. If the test runs and passes or if the test runs and fails (testFailedThisMonitoringCycle = 1) then the bit shall be set to a 0 (and latched). Reset to 1 after a call to ClearDiagnosticInformation. 1 = DTC test has not run to completion since the last time diagnostic information was cleared. 0 = DTC test has returned either a passed or failed test result at least one time since the last time diagnostic information was cleared.	C_2	C_2	TNCSLC
5	**testFailedSinceLastClear** This bit shall indicate whether a DTC test has ever returned a testFailedThisMonitoringCycle = 1 result since the last time a call was made to ClearDiagnosticInformation (latched testFailedThisMonitoringCycle =1). Zero (0) shall indicate that the test has not run or that the DTC test ran and passed (but never failed). If the test runs and fails then the bit shall remain latched at a 1. Reset to 0 after a call to ClearDiagnosticInformation. 1 = DTC test returned a testFailedThisMonitoringCycle = 1 result at least once since the last time diagnostic information was cleared. 0 = DTC test has not indicated a testFailedThisMonitoringCycle = 1 result since the last time diagnostic information was cleared.	C_2	C_2	TFSLC
6	**testNotCompletedThisMonitoringCycle** This bit shall indicate whether a DTC test has ever run and completed during the current monitoring cycle (or completed during the current monitoring cycle after the last time a call was made to ClearDiagnosticInformation). One (1) shall indicate that the DTC test has not run to completion during the current monitoring cycle. If the test runs and passes or fails then the bit shall be set (and latched) to 0 until a new monitoring cycle is started. Reset to 1 after a call to ClearDiagnosticInformation. 1 = DTC test has not run to completion this monitoring cycle (or since the last time diagnostic information was cleared this monitoring cycle). 0 = DTC test has returned either a passed or testFailedThisMonitoringCycle = 1 result during the current drive cycle (or since the last time diagnostic information was cleared during the current monitoring cycle).	M	M	TNCTMC

Table D.16 — DTC status bit definitions

Bit	Description	Cvt		Mnemonic
		Emission	Non-Emission	
7	**warningIndicatorRequested** This bit shall report the status of any warning indicators associated with a particular DTC. Warning outputs may consist of indicator lamp(s), displayed text information, etc. If no warning indicators exist for a given system or particular DTC, this status shall default to a logic ”0” state. Conditions for activating the warning indicator shall be defined by the vehicle manufacturer / implementation, but if the warning indicator is on for a given DTC, then confirmedDTC shall be also be set to 1. Reset to a logical ‘0’ after a call to ClearDiagnosticInformation. Additional reset conditions defined by vehicle manufacturer / implementation. 1 = Warning indicator requested to be ON. 0 = Warning indicator requested to be OFF.	M	U	WIR
C_1 :	Bit 1 (testFailedThisMonitoringCycle) is Mandatory if Bit 2 (pendingDTC) is supported. Bit 1 (testFailedThisMonitoringCycle) is User Optional if Bit 2 (pendingDTC) is not supported.			
C_2 :	Bit 4 (testNotPassedSinceLastClear) and Bit 5 (testNotFailedSinceLastClear) shall always be supported together.			

D.5.1 Example for operation of DTC Status Bits

The following example provides an overview on the operation of the DTC status bits. The figure shows the handling for a 3 trip OBD DTC.

The handling can also be applied to non-OBD DTCs and is shown here for general informational purpose.

NOTE In this example, the OBD server starts a monitoring cycle when the engine is started. The monitoring cycle ends (and the next monitoring cycle begins) the next time that the engine is started.

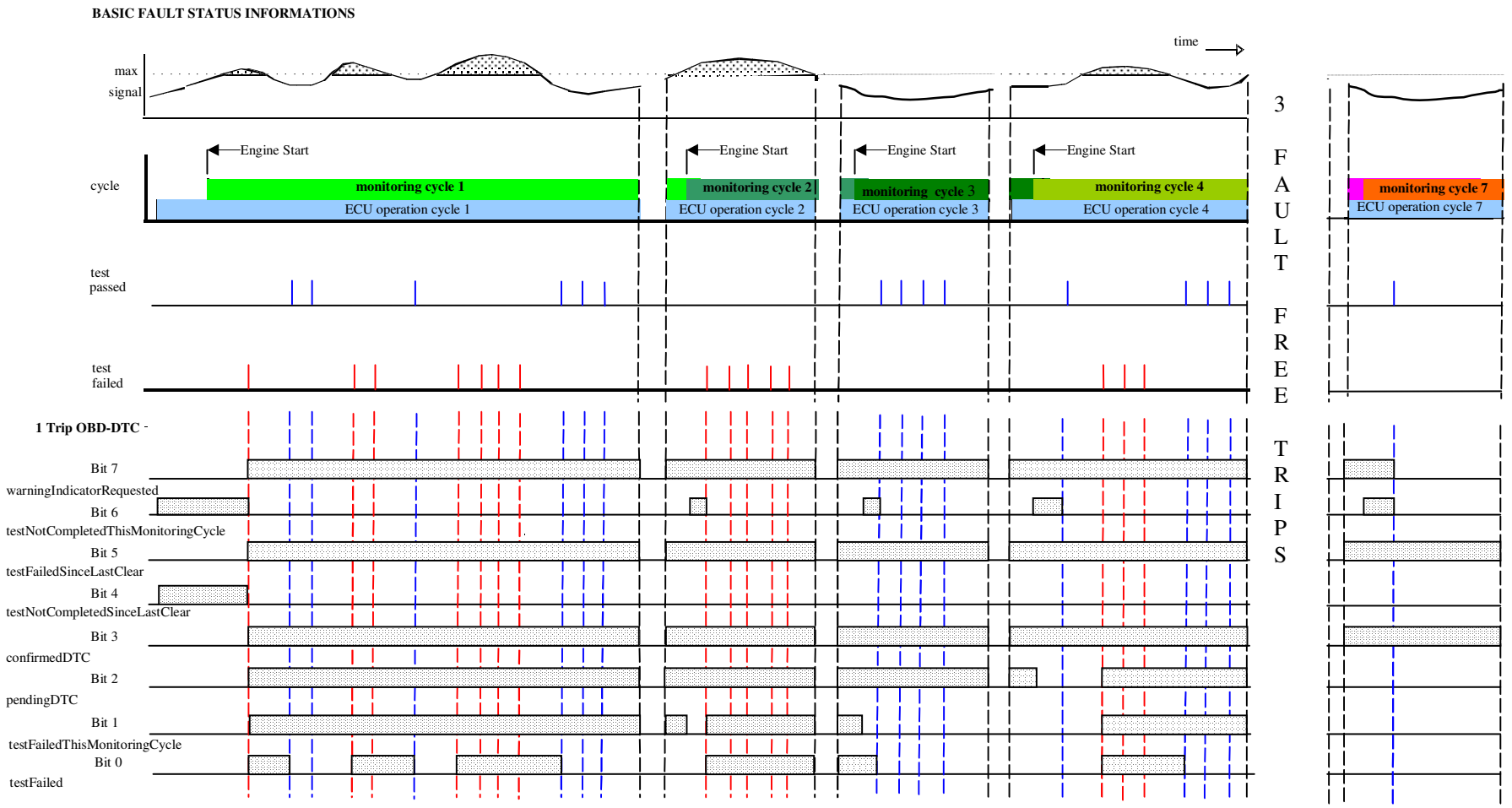


Figure D.1 — DTC Status example - timing flow for a 3-trip OBD DTC

Annex E
(normative)

Input output control functional unit data parameter definitions

E.1 InputOutputControlParameter definitions

Table E.1 — inputOutputControlParameter definitions

Hex	Description	Cvt	Mnemonic
00	**returnControlToECU** This value shall indicate to the server that the client does no longer have control about the input signal, internal parameter or output signal referenced by the inputOutputLocalIdentifier. Number of controlState bytes in request: 0 Number of controlState bytes in pos. response: depends on the inputOutputIdentifier	U	RCTECU
01	**resetToDefault** This value shall indicate to the server that it is requested to reset the input signal, internal parameter or output signal referenced by the inputOutputLocalIdentifier to its default state. Number of controlState bytes in request: 0 Number of controlState bytes in pos. response: depends on the inputOutputIdentifier	U	RTD
02	**freezeCurrentState** This value shall indicate to the server that it is requested to freeze the current state of the input signal, internal parameter or output signal referenced by the inputOutputLocalIdentifier. Number of controlState bytes in request: 0 Number of controlState bytes in pos. response: depends on the inputOutputIdentifier	U	FCS
03	**shortTermAdjustment** This value shall indicate to the server that it is requested to adjust the input signal, internal parameter or output signal referenced by the inputOutputLocalIdentifier in RAM to the value(s) included in the controlOption parameter(s). (e.g. set Idle Air Control Valve to a specific step number, set pulse width of valve to a specific value/duty cycle). Number of controlState bytes in request: depends on the inputOutputIdentifier Number of controlState bytes in pos. response: depends on the inputOutputIdentifier	U	STA
04 - FF	**reservedByDocument** This value is reserved by this document for future definition.	M	RBD

Annex F
(normative)

Remote activation of routine functional unit data parameter definitions

F.1 RoutineIdentifier definition

Table F.1 — routineIdentifier definition

Hex	Description	Cvt	Mnemonic
0000 - 00FF	**reservedByDocument** This value shall be reserved by this document for future definition.	M	RBD
0100 - 01FF	**OBDTestIds** This range of values is reserved to represent OBD/EOBD test result values.	U	RI_
0200 - EFFF	**vehicleManufacturerSpecific** This range of values is reserved for vehicle manufacturer specific use.	U	VMS
F000 - FEFF	**systemSupplierSpecific** This range of values is reserved for system supplier specific use.	U	SSS
FF00	**eraseMemory** This value shall be used to start the servers memory erase routine. The Control option and status record format shall be ECU specific and defined by the vehicle manufacturer.	U	EM
FF01	**checkProgrammingDependencies** This value shall be used to check the server's memory programming dependencies. The Control option and status record format shall be ECU specific and defined by the vehicle manufacturer.	U	CPD
FF02 - FFFF	**ReservedByDocument** This value shall be reserved by this document for future definition.	M	RBD

Annex G
(informative)

SecurityAccess service delay timer example values

G.1 SecurityAccess service delay timer example values

Each of the following tables shows the realtionship between the average number of days to gain security access in a random fashion with relation to the number of seed bytes and timeout values between successive invalid security access attempts.

Table G.1 — Delay timer for an average time to gain access of 4 days

Number of Seed Bytes	Average number of failed combinations before calculating the correct key in a random fashion	Time duration required to allow another valid seed request after a failed key submission (in seconds)
1	128	2700
2	32768	10,546875
3	8388608	0,04119873
4	2147483648	0,000160933

Table G.2 — Delay timer for an average time to gain access of 7 days

Number of Seed Bytes	Average number of failed combinations before calculating the correct key in a random fashion	Time duration required to allow another valid seed request after a failed key submission (in seconds)
1	128	4725
2	32768	18,45703125
3	8388608	0,072097778
4	2147483648	0,000281632

Table G.3 — Delay timer for an average time to gain access of 14 days

Number of Seed Bytes	Average number of failed combinations before calculating the correct key in a random fashion	Time duration required to allow another valid seed request after a failed key submission (in seconds)
1	128	9450
2	32768	36,9140625
3	8388608	0,144195557
4	2147483648	0,000563264

Table G.4 — Delay timer for an average time to gain access of 21 days

Number of Seed Bytes	Average number of failed combinations before calculating the correct key in a random fashion	Time duration required to allow another valid seed request after a failed key

Table G.4 — Delay timer for an average time to gain access of 21 days

		submission (in seconds)
1	128	14175
2	32768	55,37109375
3	8388608	0,216293335
4	2147483648	0,000844896

Table G.5 — Delay timer for an average time to gain access of 28 days

Number of Seed Bytes	Average number of failed combinations before calculating the correct key in a random fashion	Time duration required to allow another valid seed request after a failed key submission (in seconds)
1	128	18900
2	32768	73,828125
3	8388608	0,288391113
4	2147483648	0,001126528

Table G.6 — Delay timer for an average time to gain access of 100 days

Number of Seed Bytes	Average number of failed combinations before calculating the correct key in a random fashion	Time duration required to allow another valid seed request after a failed key submission (in seconds)
1	128	67500
2	32768	263,671875
3	8388608	1,029968262
4	2147483648	0,004023314

Table G.7 — Delay timer for an average time to gain access of 200 days

Number of Seed Bytes	Average number of failed combinations before calculating the correct key in a random fashion	Time duration required to allow another valid seed request after a failed key submission (in seconds)
1	128	135000
2	32768	527,34375
3	8388608	2,059936523
4	2147483648	0,008046627

Annex H
(informative)

Examples for addressAndLengthFormatIdentifier parameter values

H.1 addressAndLengthFormatIdentifier example values

The following table contains examples of combinations of values for the high and low nibble of the addressAndLengthFormatIdentifier. The following needs to be considered:

— Values, which are either marked as "not applicable" for the "manageable memorySize" or the "memoryAddress range", are not allowed to be used and have to be rejected by the server via a negative response message.

— Values with an applicable "manageable memorySize" and "memoryAddress range" are allowed for this parameter

Table H.1 — addressAndLengthFormatIdentifier example

Hex	Description			
	bit 7-4 (high nibble) number of memorySize bytes		bit 3-0 (low nibble) number of memoryAddress bytes	
	bytes used for memorySize parameter	manageable size	bytes used for memoryAddress parameter	addressable memory
00	not applicable	not applicable	not applicable	not applicable
01	not applicable	not applicable	1	256 Byte
02	not applicable	not applicable	2	64 KB
03	not applicable	not applicable	3	16 MB
04	not applicable	not applicable	4	4 GB
05	not applicable	not applicable	5	1.024 GB
06 … 0F	:	:	:	:
10	1	256 Byte	not applicable	not applicable
11	1	256 Byte	1	256 Byte
12	1	256 Byte	2	64 KB
13	1	256 Byte	3	16 MB
14	1	256 Byte	4	4 GB
15	1	256 Byte	5	1.024 GB
16 … 1F	:	:	:	:
20	2	64 KB	not applicable	not applicable
21	2	64 KB	1	256 Byte
22	2	64 KB	2	64 KB
23	2	64 KB	3	16 MB
24	2	64 KB	4	4 GB
25	2	64 KB	5	1.024 GB
26 … 2F	:	:	:	:

Table H.1 — addressAndLengthFormatIdentifier example

Hex	Description			
	bit 7-4 (high nibble) number of memorySize bytes		bit 3-0 (low nibble) number of memoryAddress bytes	
	bytes used for memorySize parameter	manageable size	bytes used for memoryAddress parameter	addressable memory
30	3	16 MB	not applicable	not applicable
31	3	16 MB	1	256 Byte
32	3	16 MB	2	64 KB
33	3	16 MB	3	16 MB
34	3	16 MB	4	4 GB
35	3	16 MB	5	1.024 GB
36 … 3F	reserved	reserved	reserved	reserved
40	4	4 GB	not applicable	not applicable
41	4	4 GB	1	256 Byte
42	4	4 GB	2	64 KB
43	4	4 GB	3	16 MB
44	4	4 GB	4	4 GB
45	4	4 GB	5	1.024 GB
46 … FF	:	:	:	: