ISO TC 22/SC 3

Date: 2002-10-01

ISO/DIS 15765-3.5

ISO TC 22/SC 3/WG 1

Secretariat: FAKRA

Road vehicles — Diagnostics on controller area network (CAN) — Part 3: Implementation of diagnostic services

Élément introductif — Élément central — Partie 3: Titre de la partie

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: (40) Enquiry
Document language: E

D:\DataShared\Diagnostics\General\standard\ISO\ISO15765\ISO 15765-3.5 Implementation of diagnostic services.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 15765-3 was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 3, *Electrical and electronic equipment*.

ISO 15765 consists of the following parts, under the general title *Road vehicles — Diagnostics on controller area network (CAN)*:

— *Part 1: General information*

— *Part 2: Network layer services*

— *Part 3: Implementation of diagnostic services*

— *Part 4: Requirements for emissions-related systems*

Introduction

International Standard ISO 15765 is applicable to vehicle diagnostic systems implemented on a Controller Area Network (CAN) communication network as specified in ISO 11898.

ISO 15765 has been established in order to define common requirements for vehicle diagnostic systems implemented on a Controller Area Network (CAN) communication link as specified in ISO 11898 Road vehicles - Controller area network (CAN).

Although primarily intended for diagnostic systems, ISO 15765 has been developed to also meet requirements from other CAN based systems needing a network layer protocol.

To achieve this, it is based on the Open Systems Interconnection (OSI) Basic Reference Model in accordance with ISO/IEC 7498 and ISO/IEC 10731 which structures communication systems into seven layers. When mapped on this model, the services specified by ISO 15765 are broken into:

— diagnostic services (layer 7), specified in ISO 15765-3,

— network layer services (layers 3), specified in ISO 15765-2,

— Controller Area Network (CAN) services (layers 1 and 2), specified in ISO 11898,

in accordance with the table below.

OSI layers	Vehicle manufacturer enhanced diagnostics	Legislated OBD
Diagnostic application	User defined	ISO 15031-5
Application layer	ISO 15765-3	ISO 15031-5
Presentation layer	N/A	N/A
Session layer	ISO 15765-3	N/A
Transport layer	N/A	N/A
Network layer	ISO 15765-2	ISO 15765-4
Data link layer	ISO 11898-1	ISO 15765-4
Physical layer	User defined	ISO 15765-4

The application layer services have been defined in compliance with diagnostic services defined in ISO 14229-1 Road vehicles — Diagnostic services — Part 1: Specification and requirements and ISO 15031-5 Road vehicles — Communication between vehicle and external test equipment for emissions-related diagnostics — Part 5: Emissions-related diagnostic services, but are not limited to be used only with these international standards. ISO 15765-3 will also be compatible with most diagnostic services defined in national standards or in vehicle manufacturer specifications.

The network layer services have been defined to be independent of the physical layer implemented. A physical layer is only specified for legislated OBD. For other application areas, ISO 15765 can be used with any Controller Area Network (CAN) physical layer.

Road vehicles — Diagnostics on controller area network (CAN) — Part 3: Implementation of diagnostic services

1 Scope

Part 3: Implementation of diagnostic services of ISO 15765 specifies the implementation of the common set of diagnostic services as specified ISO 14229-1 Diagnostic services on controller area network.

This standard defines the requirements applicable to all in-vehicle servers connected to a CAN network and the external test equipment.

This International Standard does not specify any requirement for the in-vehicle CAN bus architecture. A legislated OBD compliant vehicle shall comply with the external test equipment requirements as specified in ISO 15031-4.

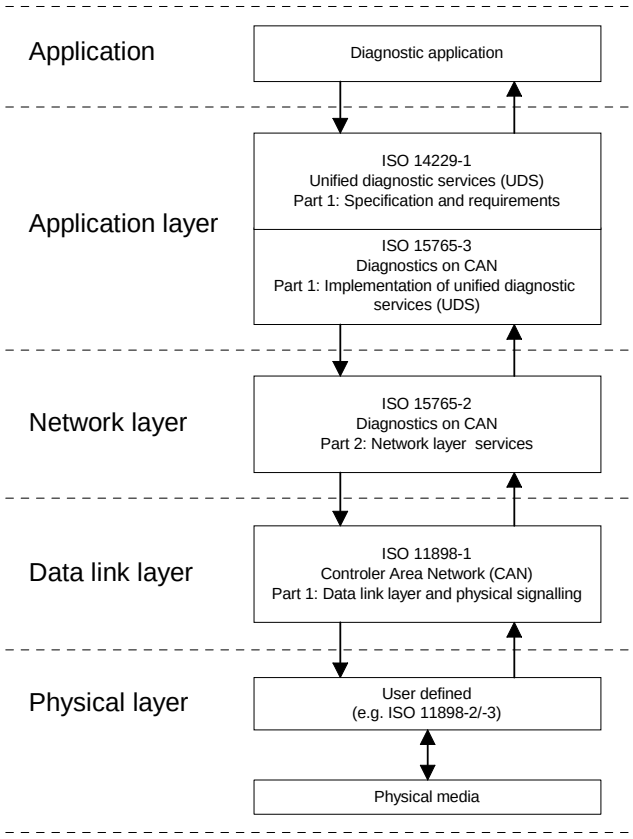

Figure 1 — Implementation of diagnostic services on CAN

2 Normative reference(s)

The following standards contain provisions, which, through reference in this text, constitute provisions of this International Standard. At the time of the publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on the International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

2.1 ISO documents

ISO 14229-1: Road vehicles — Diagnostic services — Part 1: Specification and requirements

ISO 11898-1: Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signaling

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

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

ISO 15765-1: Road vehicles — Diagnostics on controller area network (CAN) — Part 1: General information

ISO 15765-2: Road vehicles — Diagnostics on controller area network (CAN) — Part 2: Network layer service

ISO 15765-4: Road vehicles — Diagnostics on controller area network (CAN) — Part 4: Requirements for emissions-related systems

2.2 SAE publications

SAE J1939-21: Recommended Practice for Serial Control and Communications Vehicle Network - Data Link Layer. July 1998

3 Term(s) and definition(s)

For the purpose of this international standard the terms and definitions given in ISO 14229-1, ISO 15765-1, and ISO 15765-2 apply.

In addition the following terms and definitions apply:

3.1
DA
Destination Address.

3.2
ID
Identifier.

3.3
DLC
Data Length Code.

3.4
GW
Gateway.

3.5
LSB
Least Significant Bit.

3.6
MSB
Most Significant Bit.

3.7
NA
Network Address.

3.8
SA
Source Address.

3.9
SM
Subnet Mask.

3.10
TOS
Type Of Service.

4 Conventions

This international standard is based on the conventions defined in ISO 14229-1, which are guided by the conventions discussed in the O.S.I. Service Conventions (ISO/TR 8509) as they apply for diagnostic services.

5 Application and session layer

5.1 Application layer services

This international standard uses the application layer services as defined in ISO 14229-1 for client-server based systems to perform functions such as test, inspection, monitoring, diagnosis, or programming of on-board vehicle servers.

5.2 Application layer protocol

This international standard uses the application layer protocol as defined in ISO 14229-1.

5.3 Application layer and diagnostic session management timing

5.3.1 General

This section specifies the application layer and session layer timing parameters and how they are handled for the client and the server.

It has to be distinguished between the following communication scenarios:

- Physical communication during,
  - default session, and
  - non-default session – session handling required,
- Functional communication during,
  - default session, and
  - non-default session – session handling required.

For all cases the possibility of requesting an enhanced response-timing window by the server via a negative response message including a response code 78 hex shall be considered.

The network layer services as defined in ISO 15765-2 are used to perform the application layer and diagnostic session management timing in the client and the server.

NOTE Any N_USData.indication with <N_Result> not equal to N_OK that is generated in the server shall not result in a response message from the server application.

5.3.2 Application layer timing parameter definitions

The following tables contain the application layer timing parameter values for the default diagnostic session.

Table 1 — Application layer timing parameter definitions for the defaultSession

Timing Parameter	Description	Type	min	max
$P2_{CAN\_Client}$	Timeout for the client to wait after the successful transmission of a request message (indicated via N_USData.con) for the start of incoming response messages (N_USDataFirstFrame.ind of a multi-frame message or N_USData.ind of a SingleFrame	Timer reload value	$P2_{CAN\_Server\_max}$ + $\Delta P2_{CAN}$	N/A[1)]

1) The maximum time a client waits for a response message to start is left to the discretion of the client. The only requirement is that $P2_{CAN\_Client}$ must be greater than the specified minimum value of $P2_{CAN\_Client}$.

Table 1 — Application layer timing parameter definitions for the defaultSession

Timing Parameter	Description	Type	min	max
	message).			
$P2^*_{CAN\_Client}$	Enhanced timeout for the client to wait after the reception of a negative response message with response code 78 hex (indicated via N_USData.ind) for the start of incoming response messages (N_USDataFirstFrame.ind of a multi-frame message or N_USData.ind of a SingleFrame message).	Timer reload value	5000 ms + $\Delta P2_{CAN}$	N/A[2)]
$P2_{CAN\_Server}$	Performance requirement for the server to start with the response message after the reception of a request message (indicated via N_USData.ind).	Performance requirement	0	50 ms
$P2^*_{CAN\_Server}$	Performance requirement for the server to start with the response message after the transmission of a negative response message (indicated via N_USData.con) with response code 78 hex (enhanced response timing).	Performance requirement	0	5000 ms
$P3_{CAN\_Client\_Phys}$	Minimum time for the client to wait after the successful transmission of a physically addressed request message (indicated via N_USData.con) with no response required before it can transmit the next physically addressed request message (see section 5.3.5.3 for further details).	Timer reload value	$P2_{CAN\_Server\_max}$	N/A[3)]
$P3_{CAN\_Client\_Func}$	Minimum time for the client to wait after the successful transmission of a functionally addressed request message (indicated via N_USData.con) before it can transmit the next functionally addressed request message in case no response is required or the requested data is only supported by a subset of the functionally addressed servers (see section 5.3.5.3 for further details).	Timer reload value	$P2_{CAN\_Server\_max}$	N/A[4)]

The parameter $\Delta P2_{CAN}$ considers any system network design dependent delays such as delays introduced by gateways and bus bandwidth plus a safety margin (e.g. 50% of worst case). The worst-case scenario (transmission time necessary for one “round trip” from client to server and back from server to client) based on system design is impacted by:

— the number of gateways involved,

— CAN frame transmission time (baudrate),

— CAN bus utilisation,

— CAN device driver implementation method (polling vs. interrupt), and processing time of the network layer.

---

2) The value that a client uses for $P2^*_{CAN\_Client}$ is left to the discretion of the client as long as it is greater than the specified minimum value of $P2^*_{CAN\_Client}$.

3) The maximum time a client waits until it transmits the next request message is left to the discretion of the client. The only requirement is that for non-default sessions the $S3_{Server}$ timing must be kept active in the server(s).

4) The maximum time a client waits until it transmits the next request message is left to the discretion of the client. The only requirement is that for non-default sessions the $S3_{Server}$ timing must be kept active in the server(s).

The value of $\Delta P2_{CAN}$ is divided into the time to transmit the request to the addressed server and the time to transmit the response to the client.

$$\Delta P2_{CAN} = \Delta P2_{CAN\_Req} + \Delta P2_{CAN\_Rsp}$$

The following figure provides an example on how $\Delta P2_{CAN}$ can be composed.

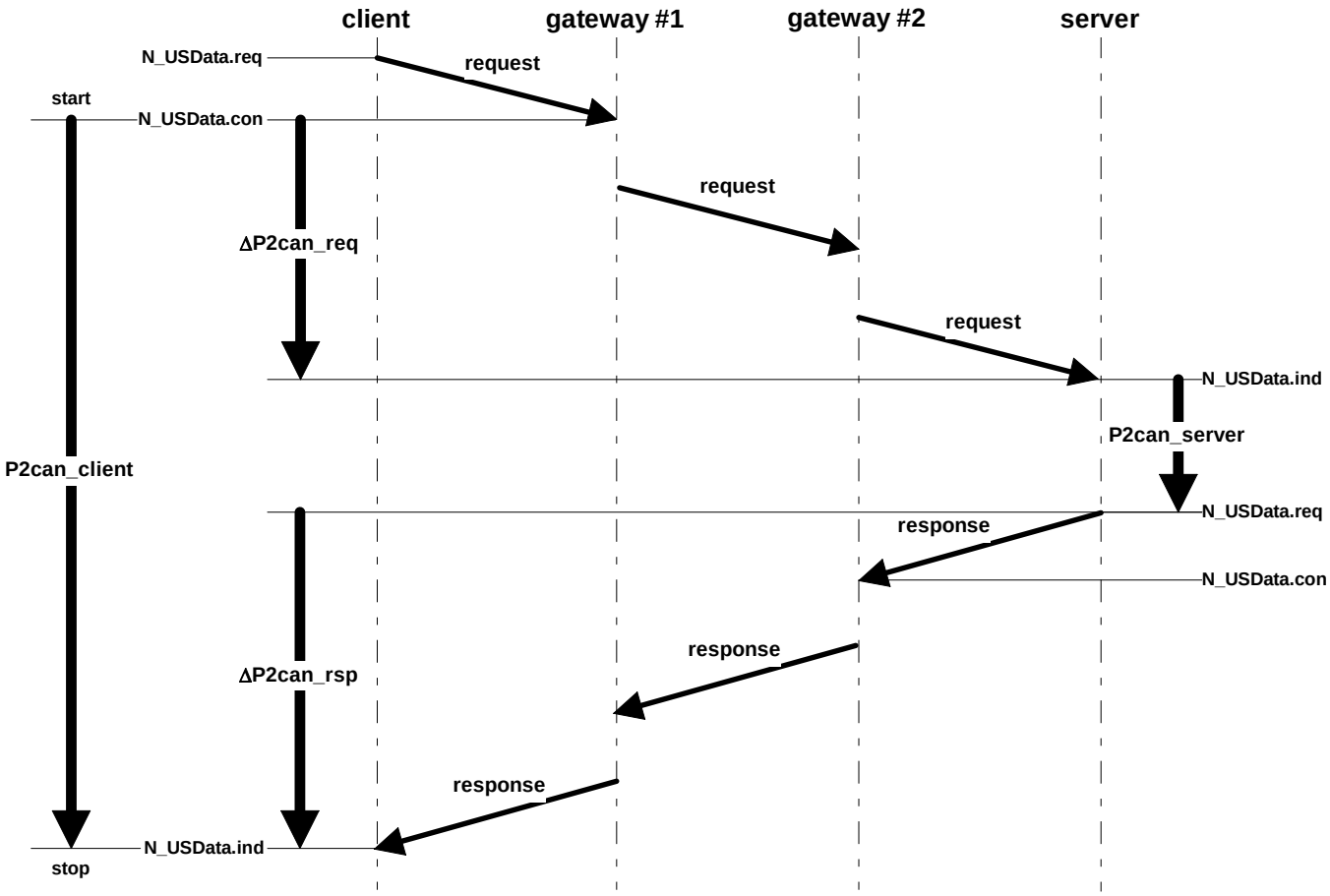

Figure 2 — Example for $\Delta P2_{Can}$ – SingleFrame request and response message

For simplicity in describing the timing parameters all further figures assume that the client and the server are located on the same network. All descriptions and figures are presented in a time related sequential order.

5.3.3 Session layer timing parameter definitions

When a diagnostic session other than the defaultSession is started then a session handling is required which is achieved via the session layer timing parameter given in the table below.

Table 2 — Session layer timing parameter definitions

Timing Parameter	Description	Type	Recommended timeout	Timeout
$S3_{Client}$	Time between functionally addressed TesterPresent (3E hex) request messages transmitted by the client to keep a diagnostic session other than the defaultSession active in multiple servers (functional communication) or maximum time between physically transmitted request messages to a single server (physical communication).	Timer reload value	2000 ms	4000 ms
$S3_{Server}$	Time for the server to keep a diagnostic session other than the defaultSession active while not receiving any diagnostic request message.	Timer reload value	N/A	5000 ms

Furthermore the server might change its application layer timings $P2_{CAN\_Server}$ and $P2^*_{CAN\_Server}$ when transitioning into a non-default session in order to achieve a certain performance or to compensate restrictions which might apply during a non-default diagnostic session. The applicable timing parameters for a non-default diagnostic session are reported in the DiagnosticSessionControl positive response message in case a response is required to be transmitted (see service description in section 8.1.1) or have to be known in advance by the client in case no response is required to be transmitted. When the client starts a non-default session functionally then it has to adapt to the timing parameters of the responding servers.

The table below defines the conditions for the client and the server to start/restart its $S3_{Client}$/$S3_{Server}$ timer. For the client it has to be distinguished between a periodically transmitted functionally addressed TesterPresent (3E hex) request message and a sequentially transmitted physically addressed TesterPresent (3E hex) request message, which is only transmitted in case of the absence of any other diagnostic request message. For the server there is no need to distinguish between that kind of TesterPresent (3E hex) handling. The table furthermore shows that the $S3_{Server}$ timer handling is based on the network layer service primitives, which means that the $S3_{Server}$ timer is also restarted upon the reception of a diagnostic request message that is not supported by the server.

Table 3 — Session layer timing start/stop conditions for the client and the server

Timing Parameter	Action	Physical and functional communication, using a functionally addressed, periodically transmitted TesterPresent request message	Physical communication only, using a physically addressed, sequentially transmitted TesterPresent request message
$S3_{Client}$	Initial start	N_USData.con that indicates the completion of the DiagnosticSessionControl (10 hex) request message.	N_USData.con that indicates the completion of the DiagnosticSessionControl (10 hex) request message in case no response is required.
			N_USData.ind that indicates the reception of the DiagnosticSessionControl (10 hex) response message in case a response is required.
	Sub-sequent start	N_USData.con that indicates the completion of the functionally addressed TesterPresent (3E hex) request message, which is transmitted each time the $S3_{Client}$ timer times out.	N_USData.con that indicates the completion of any request message in case no response is required.
			N_USData.ind that indicates the reception of any response message in case a response is required.
			N_USData.ind that indicates an error during the reception of a multi-frame response message.
$S3_{Server}$	Initial start	N_USData.con that indicates the completion of the transmission of a DiagnosticSessionControl positive response message for a transition from the default session to a non-default session, in case a response message is required.	
		Successful completion of the requested action of the service DiagnosticSessionControl (10 hex) for a transition from the default session to a non-default session, in case no response message is required/allowed.	
	Sub-sequent stop	N_USDataFirstFrame.ind that indicates the start of a multi-frame request message or N_USData.ind that indicates the reception of any SingleFrame request message.	
	Sub-sequent start	N_USData.con that indicates the completion of any response message that concludes a service execution (final response message) in case a response message is required/allowed to be transmitted (this includes positive and negative response messages). A negative response with response code 78 hex does not restart the $S3_{Server}$ timer.	
		Completion of the requested action (service conclusion) in case no response message (positive and negative) is required/allowed.	
		N_USData.ind that indicates an error during the reception of a multi-frame request message.	
		See section 5.3.5.4 for further details regarding the $S3_{Server}$ handling in the server when the server is requested to transmit unsolicited response message such as periodic data or responses based on an event.	

5.3.4 Client and server timer resource requirements

The following tables list the timer resource required for the client and the server in order to fulfil the above given timing requirements during the default session and any non-default session.

Table 4 — Timer resources requirements during defaultSession

Timing Parameter	Client	Server
$P2_{CAN\_Client}$	A single timer is required for each logical communication channel (physical and functional communication), e.g. each point-to-point communication requires a separate communication channel.	N/A
$P2_{CAN\_Server}$	N/A	An optional timer might be required for the enhanced response timing in order to make sure that sub-sequent negative response messages with response code 78 hex are transmitted prior to the expiration of $P2^*_{CAN\_Server}$.
$P3_{CAN\_Physical}$	A single timer is required per logical physical communication channel.	N/A
$P3_{CAN\_Functional}$	A single timer is required per logical functional communication channel.	N/A

During a non-default session the additional timer resource requirements as given in the following table apply for the client and the server.

Table 5 — Additional Timer resources requirements during non-defaultSession

Timing Parameter	Client	Server
$S3_{Client}$	A single timer is required when using a periodically transmitted, functionally addressed TesterPresent (3E) hex request message to keep the servers in a non-defaultSession. There is no need for additional timers per activated diagnostic sessions.	N/A
	A single timer is required for each point-to-point communication channel when using a sequentially transmitted, physically addressed TesterPresent (3E) hex request message to keep a single server in a non-defaultSession in case of the absence of another diagnostic request message then.	
$S3_{Server}$	N/A	A single timer is required in the server, because only a single diagnostic session can be active at a time in a single server.

5.3.5 Detailed timing parameter descriptions

The following sections describe in detail the certain communication scenarios and graphically depict them.

5.3.5.1 Physical communication

5.3.5.1.1 Physical communication during the defaultSession

The following figure graphically depicts the timing handling in the client and the server for a physically addressed request message during the default session. Following the figure a description can be found that references the points marked in the figure.

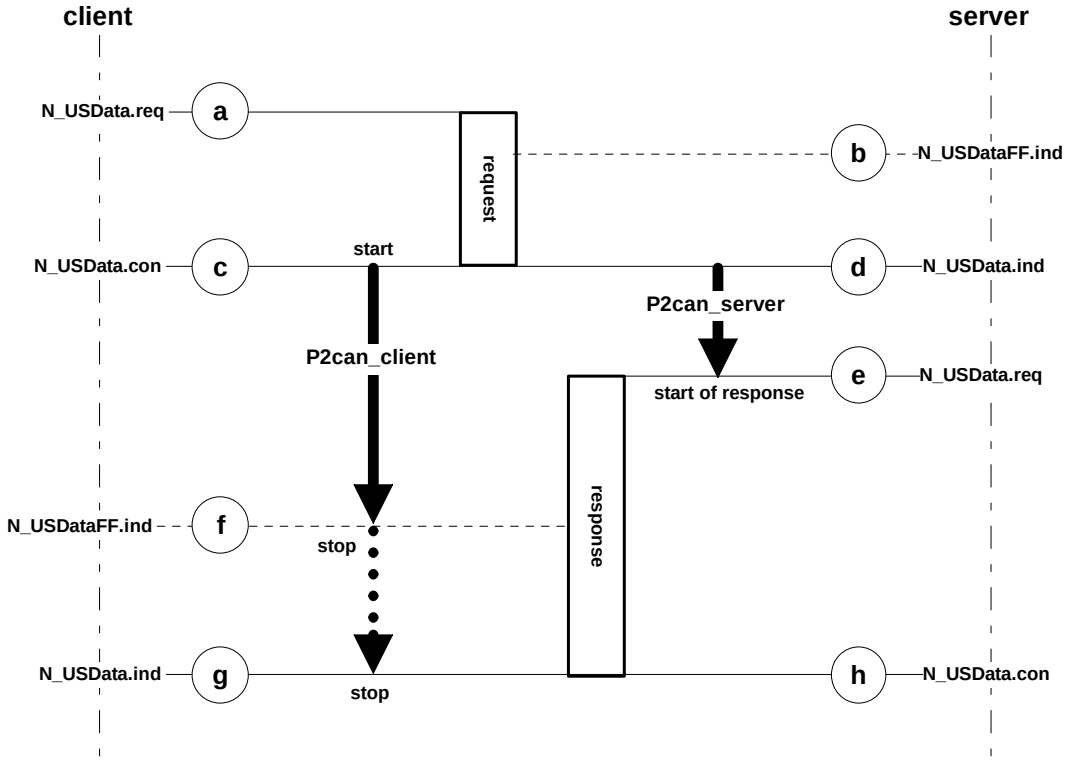

Figure 3 — Physical communication during default session

a) The diagnostic application of the client starts the transmission of the request message by issuing a N_USData.req to its network layer. The network layer transmits the request message to the server. The request message can either be a single frame message or a multi-frame message.

b) In case of a multi-frame message the start of the request is indicated in the server via N_USDataFF.ind that is issued by its network layer.

c) The completion of the request message is indicated in the client via N_USData.con. When receiving the N_USData.con the client starts its $P2_{CAN\_Client}$ timer, using the default reload value $P2_{CAN\_Client}$. The value of the $P2_{CAN\_Client}$ timer has to consider any latency that is involved based on the vehicle network design (e.g. communication over gateways, bus bandwidth, etc.). For simplicity the figure assumes that the client and the server are located on the same network.

d) The completion of the request message is indicated in the server via the N_USData.ind.

e) The server is required to start with its response message within $P2_{CAN\_Server}$ after the reception of N_USData.ind. This means that in case of a multi-frame response messages the FirstFrame has to be sent within $P2_{CAN\_Server}$ and for single frame response messages that the SingleFrame has to be sent within $P2_{CAN\_Server}$.

f) In case of a multi-frame response message the reception of the FirstFrame is indicated in the client via the N_USDataFF.ind of its network layer. When receiving the FirstFrame indication the client stops its $P2_{CAN\_Client}$ timer.

g) The network layer will generate a final N_USData.ind in case the complete message is received or an error occurred during the reception. In case of a single frame response message the reception of the SingleFrame is indicated in the client via a single N_USData.ind. When receiving this single frame indication the client stops its $P2_{CAN\_Client}$ timer.

h) The completion of the response message is indicated in the server via N_USData.con.

5.3.5.1.2 Physical communication during the defaultSession with enhanced response timing

The following figure graphically depicts the timing handling in the client and the server for a physically addressed request message during the default session and the request of the server for an enhanced response timing (negative response code 78 hex handling). Following the figure a description can be found that references the points marked in the figure.

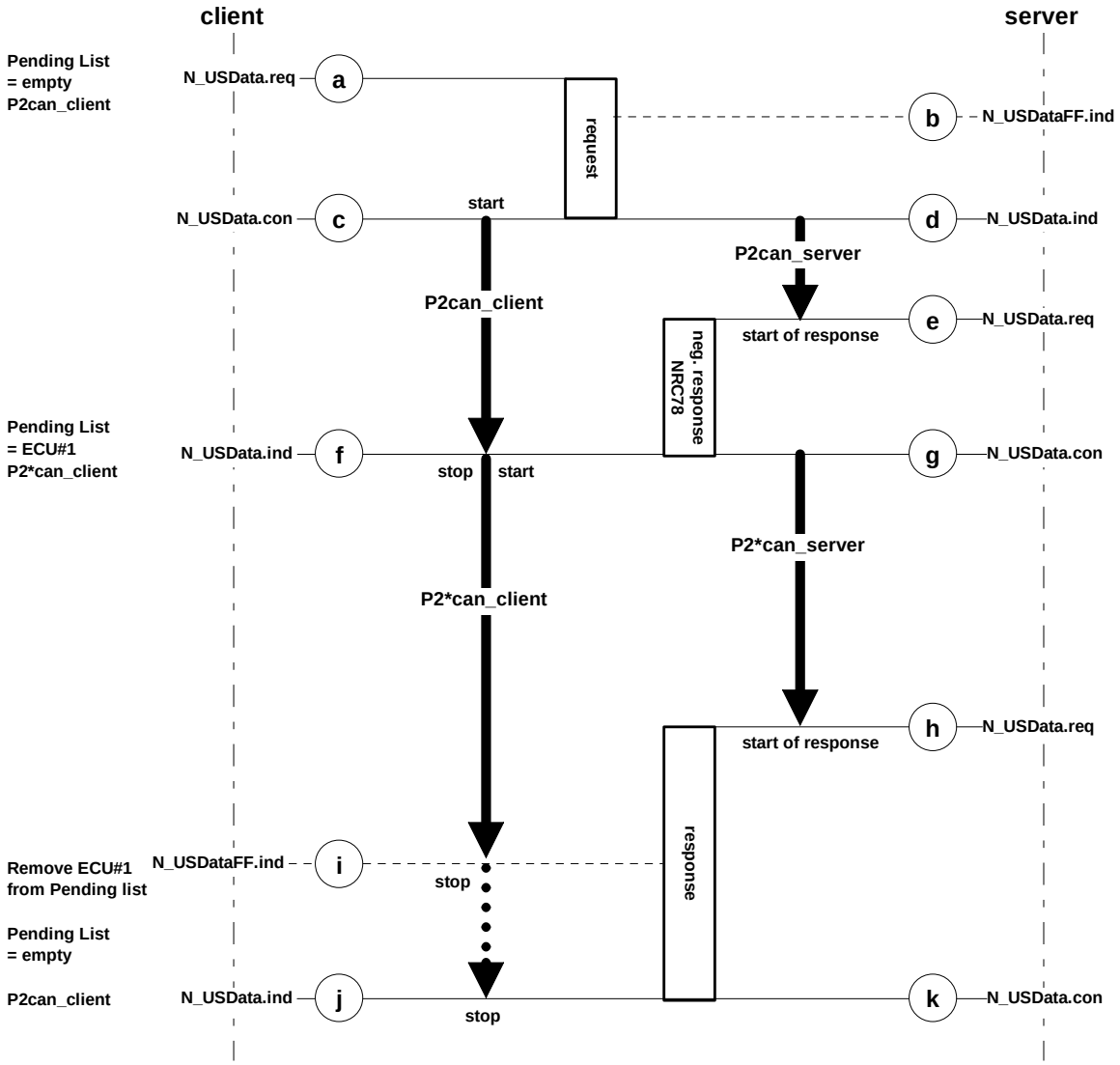

Figure 4 — Physical communication during non-default session – enhanced response timing

a) The diagnostic application of the client starts the transmission of the request message by issuing a N_USData.req to its network layer. The network layer transmits the request message to the server. The request message can either be a single frame message or a multi-frame message.

b) In case of a multi-frame message the start of the request is indicated in the server via N_USDataFF.ind that is issued by its network layer.

c) The completion of the request message is indicated in the client via N_Usdata.con. When receiving the N_USData.con the client starts its $P2_{CAN\_Client}$ timer, using the default reload value $P2_{CAN\_Client}$. The value of the $P2_{CAN\_Client}$ timer has to consider any latency that is involved based on the vehicle network design (e.g. communication over gateways, bus bandwidth, etc.). For simplicity the figure assumes that the client and the server are located on the same network.

d) The completion of the request message is indicated in the server via N_USData.ind.

e) The server is required to start with its response message within $P2_{CAN\_Server}$ after the reception of N_USData.ind. This means that in case of a multi-frame response message the FirstFrame has to be sent within $P2_{CAN\_Server}$ and for single frame response messages that the SingleFrame has to be sent within $P2_{CAN\_Server}$.

f) In case the server cannot provide the requested information within the $P2_{CAN\_Server}$ response timing it can request an enhanced response timing window by sending a negative response message including response code 78 hex. Upon the reception of the negative response message within the client, the client network layer generates a N_USData.ind. The reception of a negative response message with response code 78 hex causes the client to restart its $P2_{CAN\_Client}$ timer, but using the enhanced reload value $P2^*_{CAN\_Client}$.

g) The server is required to start with its response message within the enhanced $P2_{CAN\_Server}$ ($P2^*_{CAN\_Server}$) following the N_USData.con of the transmitted negative response message. In case the server can still not provide the requested information within the enhanced $P2^*_{CAN\_Server}$ then a further negative response message including response code 78 hex can be sent by the server. This will cause the client to restart its $P2_{CAN\_Client}$ timer again, using the enhanced reload value $P2^*_{CAN\_Client}$. For simplicity the figure below only shows a single negative response message with response code 78 hex.

h) Once the server can provide the requested information (positive response or negative response other than response code 78 hex) it starts with its final response message.

i) In case of a multi-frame final response message the reception of the FirstFrame is indicated in the client via the N_USDataFF.ind of the network layer. When receiving the FirstFrame indication the client stops its $P2_{CAN\_Client}$ timer.

j) The network layer of the client will generate a final N_USData.ind in case the complete message is received or an error occurred during the reception. In case of a single frame response message the reception of the SingleFrame is indicated in the client via a single N_USData.ind. When receiving this single frame indication the client stops its $P2_{CAN\_Client}$ timer.

k) The completion of the transmission will also be indicated in the server via N_USData.con.

5.3.5.1.3 Physical communication during a non-default session

5.3.5.1.3.1 Functionally addressed TesterPresent (3E hex) message

The following figure graphically depicts the timing handling in the client and the server when performing physical communication during a non-default session (e.g. programmingSession) and using a functionally addressed, periodically transmitted TesterPresent (3E hex) request message that does not require a response message from the server. Following the figure a description can be found that references the points marked in the figure.

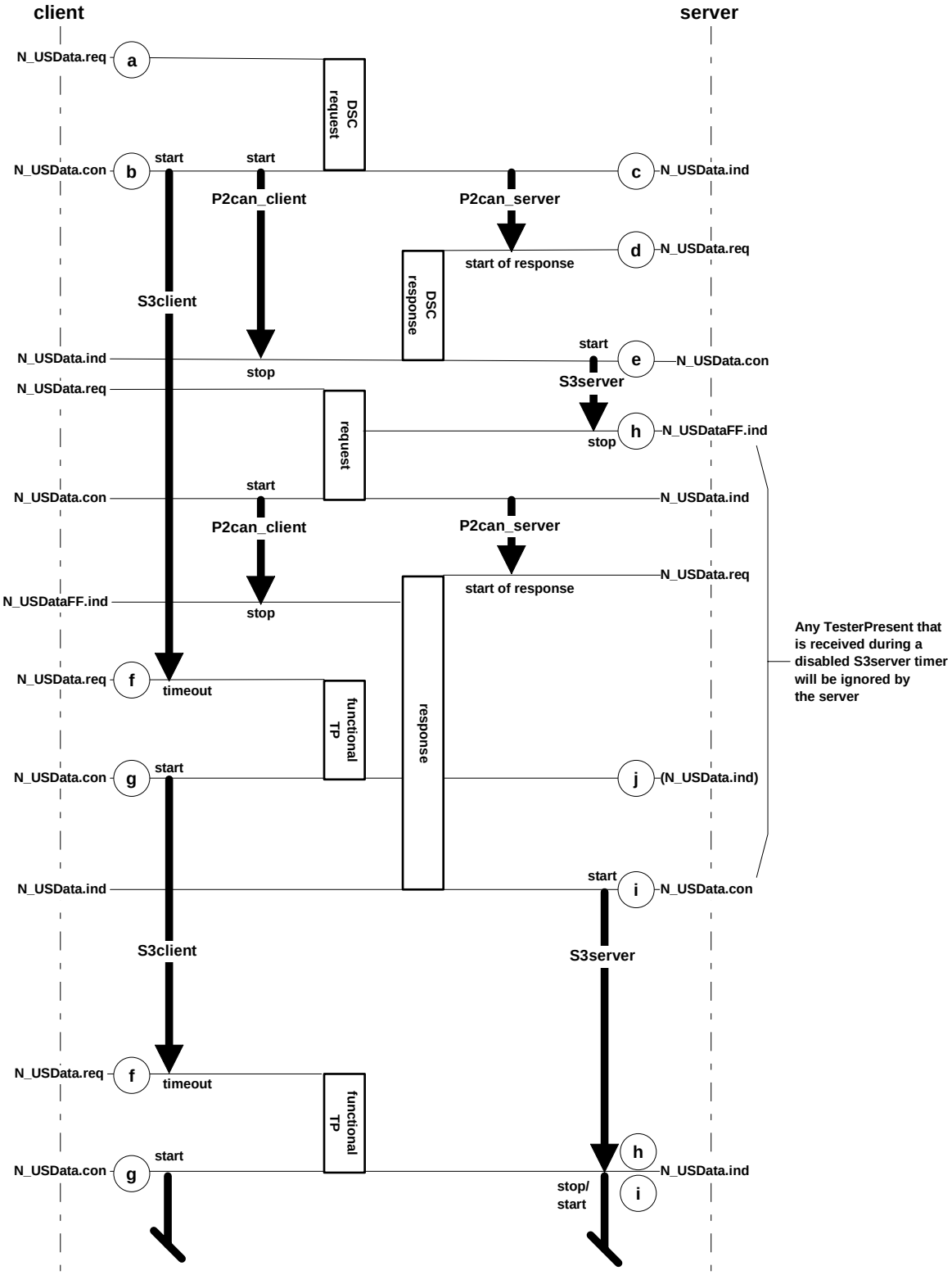

Figure 5 — Physical communication during non-default session – functionally addressed TesterPresent

The handling of the $P2_{CAN\_Client}$ and $P2_{CAN\_Server}$ timing is identical to the handling as described in the sections 5.3.5.1.1 and 5.3.5.1.2. The only exception is that the reload values on the client side and the resulting time where the server shall send its final response time might differ. This is based on the transition into a session other than the default session where different $P2_{CAN\_Client}$ timing parameters might apply (see DiagnosticSessionControl (10 hex) service in section 8.1.1 for details on how the timing parameters are reported to the client).

a) The diagnostic application of the client starts the transmission of the DiagnosticSessionControl (10 hex) request message by issuing a N_USData.req to its network layer. The network layer transmits the request message to the server.

b) The request message is a single frame message. Its completion is indicated in the client via the N_USData.con. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies. The generated N_USData.con in the client causes the start of the $S3_{Client}$ timer (session timer).

c) The completion of the request message is indicated in the server via the N_USData.ind. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies.

d) For the figure given it is assumed that the client requires a response from the server. The server has to transmit the DiagnosticSessionControl (10 hex) positive response message.

e) The completion of the transmission of the response message is indicated in the server via N_USData.con. Now the server starts its $S3_{Server}$ timer, which keeps the activated non-default session active as long as it does not time out. It is the client's responsibility to ensure that the $S3_{Server}$ timer is reset prior to its timeout to keep the server in the non-default session.

f) Once the $S3_{Client}$ timer is started in the client this causes the transmission of a functionally addressed TesterPresent (3E hex) request message, which does not require a response message, each time the $S3_{Client}$ timer times out.

g) Upon the indication of the completed transmission of the TesterPresent (3E hex) request message via N_USData.con of its network layer the client once again starts its $S3_{Client}$ timer. This means that the functionally addressed TesterPresent (3E hex) request message is sent on a periodic basis every time $S3_{Client}$ times out.

h) Any time the server is in the process of handling any diagnostic service it stops its $S3_{Server}$ timer.

i) When the diagnostic service is completely processed then the server restarts its $S3_{Server}$ timer. This means that any diagnostic service, including TesterPresent (3E hex), resets the $S3_{Server}$ timer. A diagnostic service is meant to be in progress any time between the start of the reception of the request message (N_USDataFF.ind or N_USData.ind receive) and the completion of the transmission of the response message in case a response message is required or the completion of any action that is caused by the request in case no response message is required (point in time reached that would cause the start of the response message). This includes negative response messages including response code 78 hex.

j) Any TesterPresent (3E hex) request message that is received during processing another request message can be ignored by the server, because it has already stopped its $S3_{Server}$ timer and will restart it once the service that is in progress is processed completely.

5.3.5.1.3.2 Physically addressed TesterPresent (3E hex) message

The following figure graphically depicts the timing handling in the client and the server when performing physical communication during a non-default session (e.g. programmingSession) and using a physically addressed TesterPresent (3E hex) request message that requires a response message from the server to keep the diagnostic session active in case of the absence of any other diagnostic service. Following the figure a description can be found that references the points marked in the figure.

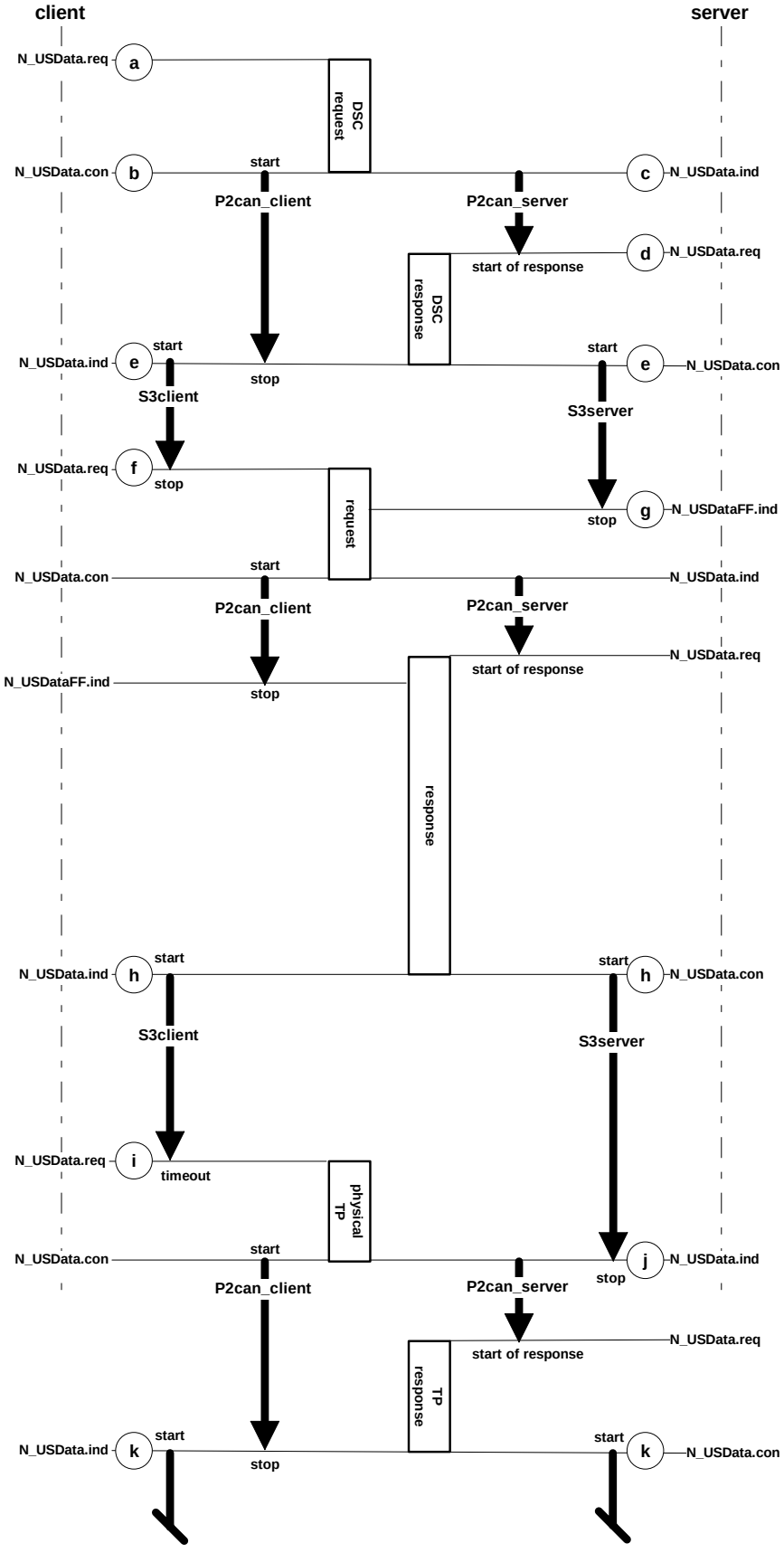

Figure 6 — Physical communication during non-default session – physically addressed TesterPresent

a) The diagnostic application of the client starts the transmission of the DiagnosticSessionControl (10 hex) request message by issuing a N_USData.req to its network layer. The network layer transmits the request message to the server.

b) The request message is a single frame message. Its completion is indicated in the client via the N_USData.con. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies. The generated N_USData.con (b) in the client does not cause the start of the $S3_{Client}$ timer (session timer), as it would for the case of using a functionally addressed and periodically transmitted TesterPresent (3E hex) message to keep a diagnostic session alive (see 5.3.5.1.3.1).

c) The completion of the request message is indicated in the server via the N_USData.ind. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies.

d) For the figure given it is assumed that the client requires a response from the server. The server has to transmit the DiagnosticSessionControl (10 hex) positive response message.

e) The completion of the transmission of the response message is indicated in the server via N_USData.con. Now the server starts its $S3_{Server}$ timer, which keeps the activated non-default session active as long as it does not time out. In the client the reception of the DiagnosticSessionControl (10 hex) positive response message is indicated via N_USData.ind. This causes the start of the $S3_{Client}$ timer. It is the client's responsibility to ensure that the $S3_{Server}$ timer is reset prior to its timeout to keep the server in the non-default session.

f) Whenever the client transmits a request message to the server (including the TesterPresent (3E hex) message) it stops its $S3_{Client}$ timer.

g) The reception of either a SingleFrame or a FirstFrame of the request message stops the $S3_{Server}$ timer in the server. The completion of the request message is indicated in the server via N_USData.ind. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies.

h) The completion of the response message is indicated in the client via N_USData.ind, which causes the client to start its $S3_{Client}$. The completion of the response message is indicated in the server via N_USData.con, which causes the server to start its $S3_{Server}$. In case the client would not require a response message than it has to start its $S3_{Client}$ timer when it receives the confirmation of the completion of the request message, which is indicated via N_USData.con. The server would start its $S3_{Server}$ timer when it has completed the requested action. For simplicity the figure shows that a response is required.

i) In case the client would not send any diagnostic request message prior to the timeout of $S3_{Client}$ then the timeout of the $S3_{Client}$ timer causes the client to transmit a physically addressed TesterPresent (3E hex) request message.

j) The reception of the TesterPresent (3E hex) request message is indicated in the server via N_USData.ind. This causes the server to stop its $S3_{Server}$ timer. . Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies.

k) The completion of the TesterPresent (3E hex) response message is indicated in the client via N_USData.ind, which causes the client to start its $S3_{Client}$. The completion of the TesterPresent (3E hex) response message is indicated in the server via N_USData.con, which causes the server to start its $S3_{Server}$. In case the client would not require a response message than it has to start its $S3_{Client}$ timer when it receives the confirmation of the completion of the TesterPresent (3E hex) request message, which is indicated via N_USData.con. The server would start its $S3_{Server}$ timer when it has completed the requested action. For simplicity the figure shows that a response is required.

5.3.5.2 Functional communication

5.3.5.2.1 Functional communication during the defaultSession

The following figure graphically depicts the timing handling in the client and two (2) servers for a functionally addressed request message during the default session. Following the figure a description can be found that references the points marked in the figure.

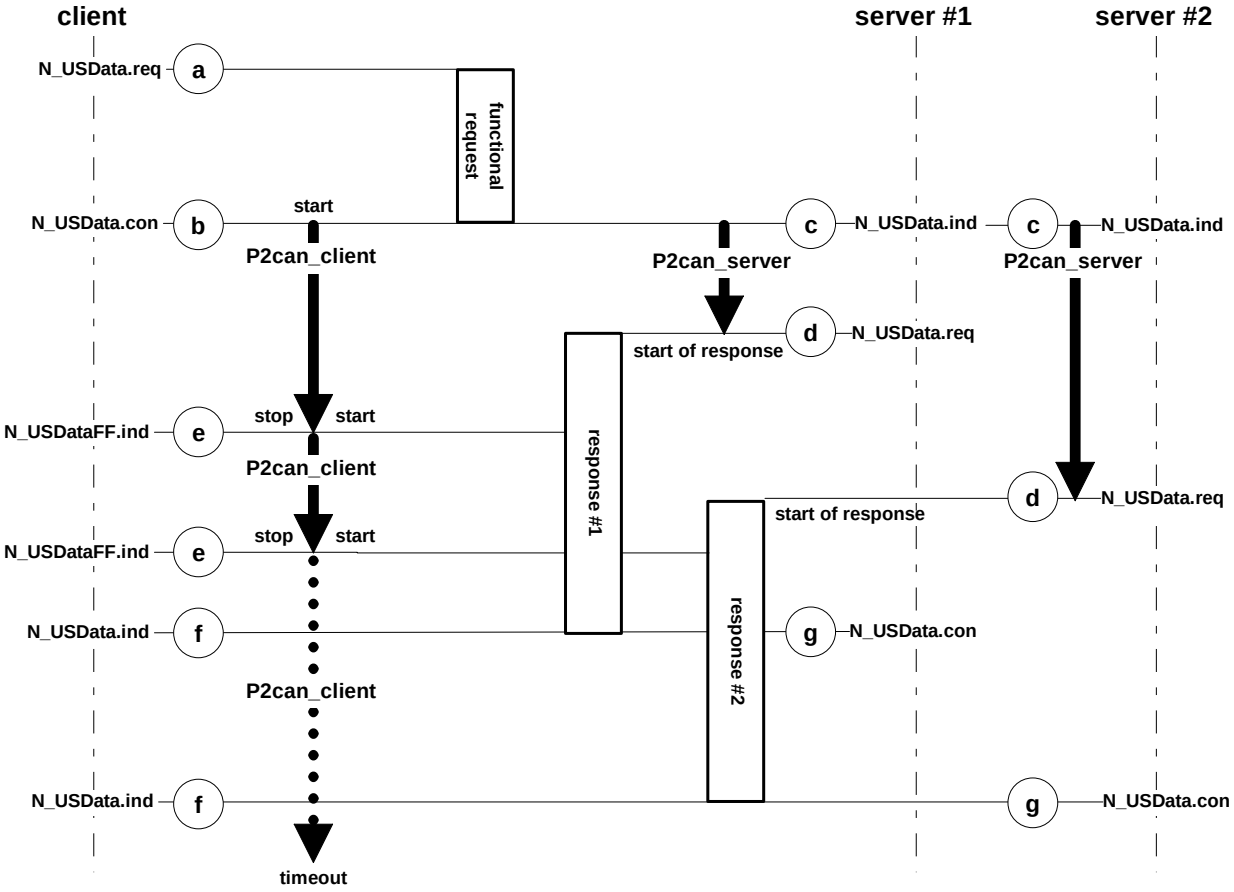

Figure 7 — Functional communication during default session

From a server point of view there is no difference in the timing handling compared to a physically addressed request message, but the client has to handle the timing different compared to physical communication.

a) The diagnostic application of the client starts the transmission of a functionally addressed request message by issuing a N_USData.req to its network layer. The network layer transmits the request message to the servers. A functionally addressed request message shall only be a single frame message.

b) The completion of the request message is indicated in the client via N_Usdata.con. When receiving the N_USData.con the client starts its $P2_{CAN\_Client}$ timer, using the default reload value $P2_{CAN\_Client}$. As for physical communication the value of the $P2_{CAN\_Client}$ timer has to consider any latency that is involved based on the vehicle network design (e.g. communication over gateways, bus bandwidth, etc.). For simplicity the figure assumes that the client and the server are located on the same network.

c) The completion of the request message is indicated in the servers via N_USData.ind.

d) The functionally addressed servers are required to start with their response messages within $P2_{CAN\_Server}$ after the reception of N_USData.ind. This means that in case of a multi-frame response messages the FirstFrame has to be sent within $P2_{CAN\_Server}$ and for single frame response messages that the SingleFrame has to be sent within $P2_{CAN\_Server}$.

e) In case of a multi-frame response message the reception of the FirstFrame from any server is indicated in the client via the N_USDataFF.ind of the network layer. A single frame response message is indicated via N_USData.ind.

f) When receiving the FirstFrame/SingleFrame indication of an incoming response message the client either stops its $P2_{CAN\_Client}$ in case it knows the servers to be expected to respond and all servers have responded or it restarts its $P2_{CAN\_Client}$ timer in case not all expected servers responded yet or the client does not know the servers to be expected to respond (client awaits the start of further response messages). The network layer of the client will generate a final N_USData.ind in case the complete message is received or an error occurred during the reception. The reception of a final N_USData.ind of a multi-frame message in the client will not have any influence on the $P2_{CAN\_Client}$ timer.

g) The completion of the transmission of the response message will also be indicated in the servers via N_USData.con.

5.3.5.2.2 Functional communication during defaultSession with enhanced response timing

The following figure graphically depicts the timing handling in the client and two (2) servers for a functionally addressed request message during the default session, where one server requests an enhanced response timing via a negative response message including response code 78 hex. Following the figure a description can be found that references the points marked in the figure.

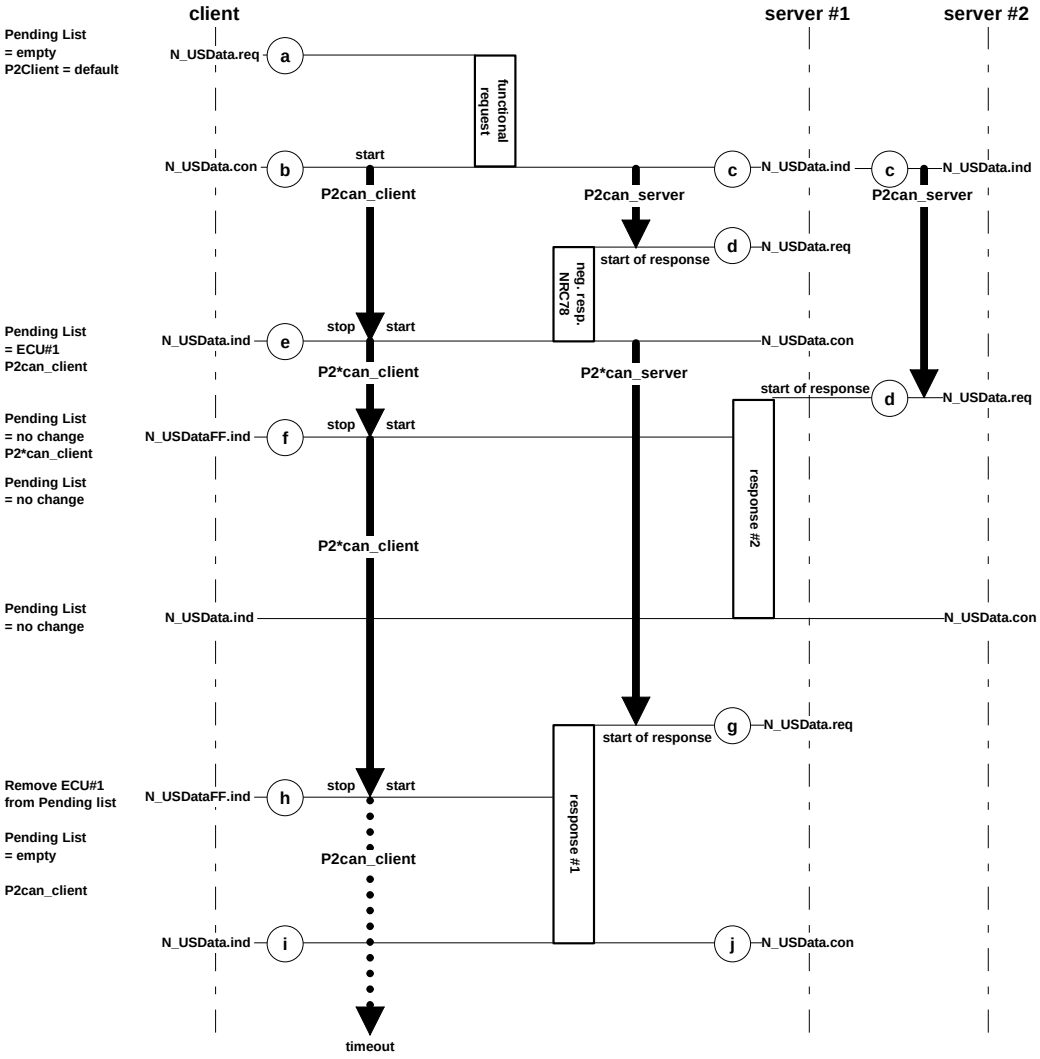

Figure 8 — Functional communication during default session – enhanced response timing

From a server point of view there is no difference in the timing handling compared to a physically addressed request message that requires enhanced response timing, but the client has to handle the timing differently compared to physical communication.

a) The diagnostic application of the client starts the transmission of the functionally addressed request message by issuing a N_USData.req to its network layer. The network layer transmits the request message to the servers. A functionally addressed request message shall only be a single frame message.

b) The completion of the request message is indicated in the client via N_USData.con. When receiving N_USData.con the client starts its $P2_{CAN\_Client}$ timer, using the default reload value $P2_{CAN\_Client}$. As for physical communication the value of the $P2_{CAN\_Client}$ timer has to consider any latency that is involved based on the vehicle network design (e.g. communication over gateways, bus bandwidth, etc.). For simplicity the figure assumes that the client and the server are located on the same network.

c) The completion of the request message is indicated in the servers via N_USData.ind.

d) The functionally addressed servers are required to start with their response messages within $P2_{CAN\_Server}$ after the reception of N_USData.ind. This means that in case of a multi-frame response messages the FirstFrame has to be sent within $P2_{CAN\_Server}$ and for single frame response messages that the SingleFrame has to be sent within $P2_{CAN\_Server}$. In case any of the addressed servers cannot provide the requested information within the $P2_{CAN\_Server}$ response timing it can request an enhanced response timing window by sending a negative response message including response code 78 hex.

e) Upon the reception of the negative response message within the client the client network layer generates a N_USData.ind. The reception of a negative response message with response code 78 hex causes the client to restart its $P2_{CAN\_Client}$ timer, using the enhanced reload value $P2^*_{CAN\_Client}$. In addition the client has to store a server identification in a list of pending response messages. Once a server that is stored as pending in the client starts with its final response message (positive response message or negative response message including a response code other than 78 hex) it is deleted from the list of pending response message. In case no further response messages are pending the client re-uses the default reload value for its $P2_{CAN\_Client}$ timer. For simplicity the figure below only shows a single negative response message including response code 78 hex from server #1.

f) As long as there is at least one server stored as pending in the client any start of a further response message from any server that was addressed by the request will cause a restart of the $P2_{CAN\_Client}$ timer using the enhanced reload value $P2^*_{CAN\_Client}$. In the figure given below this is shown when the client receives the start of the response message of the second server.

g) As for physical communication the server who requested enhanced response timing is required to start with its response message within the enhanced $P2_{CAN\_Server}$ ($P2^*_{CAN\_Server}$). Once the server can provide the requested information it starts with its final response message by issuing a N_USData.req to its network layer. When the server can still not provide the requested information within the enhanced $P2^*_{CAN\_Server}$ then a further negative response message including response code 78 hex can be sent. This will cause the client to restart its $P2_{CAN\_Client}$ timer again, using the enhanced reload value $P2^*_{CAN\_Client}$. A negative response message including response code 78 hex from a server that is already stored in the list of pending response messages has no affect to the client internal list of pending response message.

h) As described in section 5.3.5.2.1 in case of a multi-frame response message the reception of the FirstFrame from any server is indicated in the client via the N_USDataFF.ind of the network layer. A single frame response message is indicated via N_USData.ind. When receiving the FirstFrame/SingleFrame indication of an incoming response message the client either stops its $P2_{CAN\_Client}$ in case it knows the servers to be expected to respond and all servers have responded or it restarts its $P2_{CAN\_Client}$ timer in case not all expected servers responded yet or the client does not know the servers to be expected to respond (client awaits the start of further response messages).

i) The network layer will generate a final N_USData.ind in case any multi-frame response message is completely received or an error occurred during the reception. This will not have any influence on the $P2_{CAN\_Client}$ timer. Furthermore the handling of the list of pending response messages as described above applies.

j) The completion of the transmission will also be indicated in the servers via N_USData.con.

5.3.5.2.3 Functional communication during non-default session

The following figure graphically depicts the timing handling in the client and two (2) servers for a functionally addressed request message during the non-default session (e.g. programmingSession), where one server requests an enhanced response timing via a negative response message including response code 78 hex. Following the figure a description can be found that references the points marked in the figure.

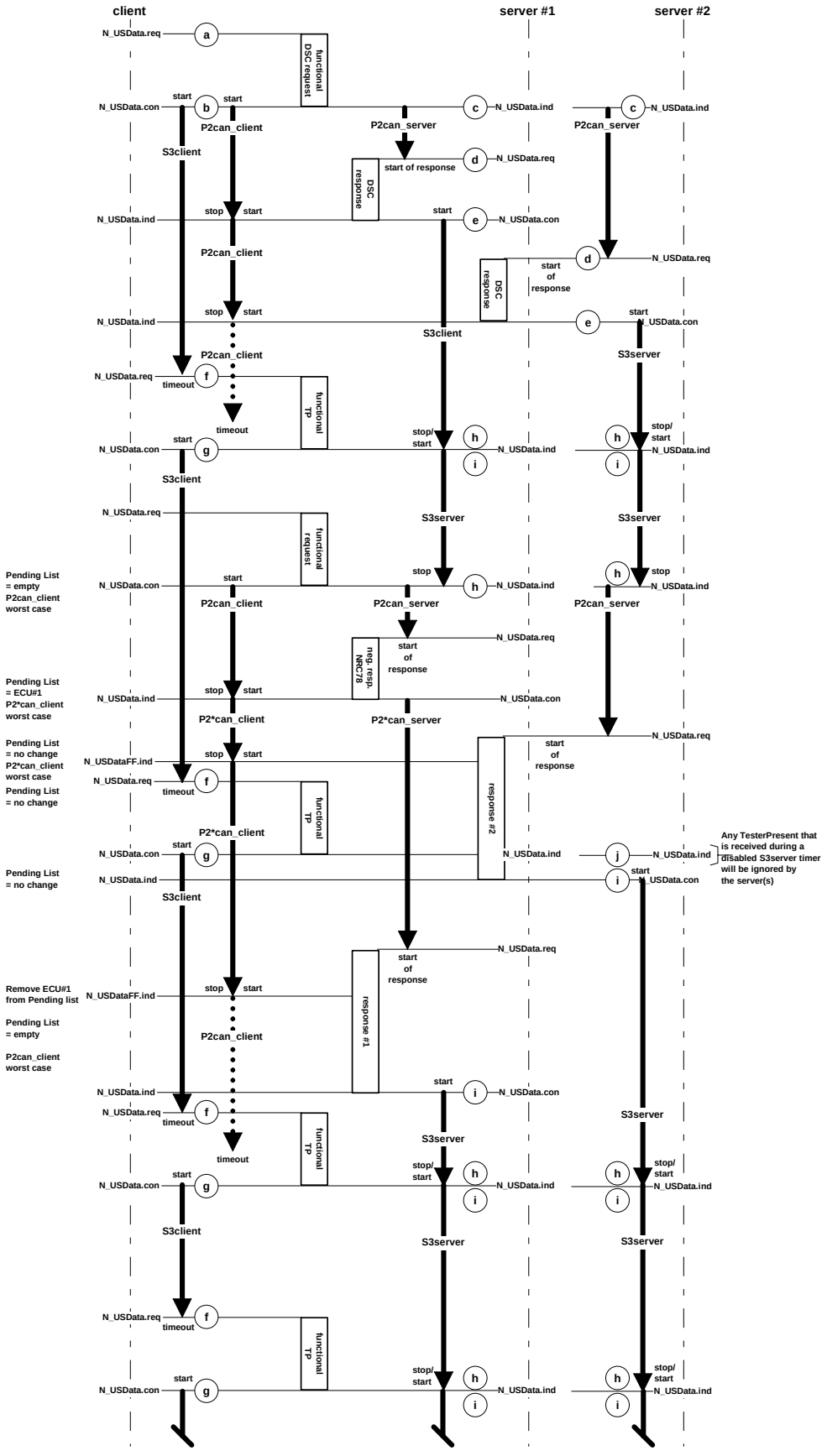

Figure 9 — Functional communication during non-default session

The handling of the $P2_{CAN\_Client}$ and $P2_{CAN\_Server}$ timing is identical to the handling as described In section 5.3.5.2.1 and 5.3.5.2.2. The only exception is that the reload values on the client side and the resulting time the server has to send its final response time might differ. This is based on the transition into a session other than the default session where different $P2_{CAN\_Client}$ timing parameters might apply (see DiagnosticSessionControl (10 hex) service in section 8.1.1 for details on how the timing parameters are reported to the client).

a) The diagnostic application of the client starts the transmission of the functionally addressed DiagnosticSessionControl (10 hex) request message by issuing a N_USData.req to its network layer. The network layer transmits the request message to the servers. The request message is a single frame message.

b) The completion of the request message is indicated in the client via N_USData.con. Now the response timing as described in section 5.3.5.2.1 and 5.3.5.2.2 applies. In addition the generated N_USData.con in the client causes the start of the $S3_{Client}$ timer (session timer).

c) The completion of the request message is indicated in the servers via N_USData.ind. Now the response timing as described in section 5.3.5.2.1 and 5.3.5.2.2 applies.

d) For the figure given it is assumed that the client requires a response from the servers. The servers have to transmit the DiagnosticSessionControl (10 hex) positive response messages.

e) The completion of the transmission of the positive response message is indicated in the servers via N_USData.con. The servers start their $S3_{Server}$ timers, which keeps the activated non-default session active as long as $S3_{Server}$ does not time out. It is the client's responsibility to ensure that the $S3_{Server}$ timer is reset prior to its timeout to keep the servers in the non-default session.

f) Once the $S3_{Client}$ timer is started in the client this causes the transmission of a functionally addressed TesterPresent (3E hex) request message, which does not require a response message each time the $S3_{Client}$ timer times out.

g) Upon the indication of the completed transmission of the TesterPresent (3E hex) request message via N_USData.con of its network layer the client once again starts its $S3_{Client}$ timer. This means that the functionally addressed TesterPresent (3E hex) request message is sent on a periodic basis every time $S3_{Client}$ times out.

h) Any time a server is in the process of handling any diagnostic service it stops its $S3_{Server}$ timer.

i) When the diagnostic service is completely processed then the server restarts its $S3_{Server}$ timer. A diagnostic service is meant to be in progress any time between the start of the reception of the request message (N_USDataFF.ind or N_USData.ind receive) and the completion of the transmission of the response message in case a response message is required or the completion of any action that is caused by the request in case no response message is required (point in time reached that would cause the start of the response message). This includes negative response messages with response code 78 hex.

j) Any TesterPresent (3E hex) request message that is received during processing another request message can be ignored by the server, because it has stopped its $S3_{Server}$ timer and will restart it once the other service is processed completely.

5.3.5.3 Minimum time between client request messages

The minimum time between request messages transmitted by the client is required in order to allow for e.g. a polling driven service data interpretation in the server. Based on its normal functionality a server might process diagnostic request messages with a certain scheduling rate (e.g. 10 ms). The time for the diagnostic service data interpretation scheduler has to be smaller than the performance requirement $P2_{CAN\_Server}$ in order to meet the server requirements specified in sections 5.3.5 and 5.3.5.1.3.2.

The timing parameter for the minimum time between request message is divided into the following two timing parameters

— $P3_{CAN\_Functional}$: This timing parameter applies to any functionally addressed request message, because it can be the case that a server is not required to respond to a functionally addressed request message if it does not support the requested data.

— $P3_{CAN\_Physical}$: This timing parameter applies to any physically addressed request message in case there is no response required to be transmitted by the server (responseRequired = no).

In case of physical communication where a response is required by the server the client can transmit the next request immediately after the complete reception of the last response message, because the server has responded completely to the request, which means that the request is completely handled by the server.

The following figure graphically depicts an example for a problem that can occur during functional communication when the client would transmit the next request immediately after it has determined that all expected servers responded to a previous request message. Following the figure a description can be found that references the points marked in the figure.

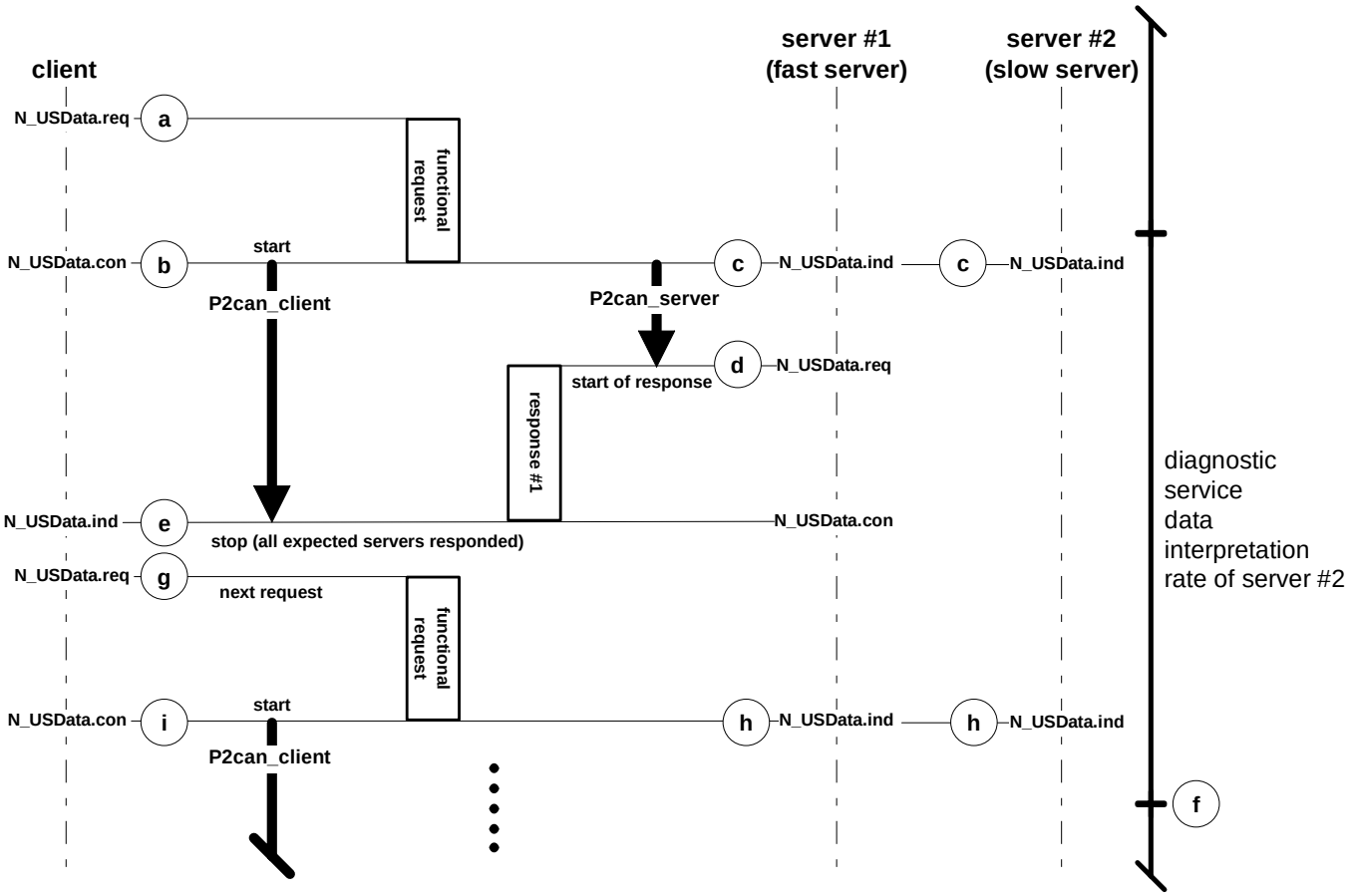

Figure 10 — Critical issue when transmitting the next request too early

a) The diagnostic application of the client starts the transmission of a functionally addressed request message by issuing a N_USData.req to its network layer. The network layer transmits the request to the servers.

b) The completion of the request message is indicated in the client via N_USData.con. The client starts its $P2_{CAN\_Client}$ timer, using the default reload value $P2_{CAN\_Client}$.

c) The completion of the request message is indicated in the server via N_USData.ind. The server starts its $P2_{CAN\_Server}$ timer, using the default reload value $P2_{CAN\_Server}$.

d) For the request message it is assumed that only server #1 supports the requested information, which means that there will be no response message from server #2. Server #1 is a fast server and can immediately process the received request message and transmits its response within $P2_{CAN\_Server}$.

e) The client receives the response message. This is indicated via N_USData.ind. The client only expected a response message from server #1 therefore it stops its timer $P2_{CAN\_Client}$.

f) Server #2 is a slow server and interprets received requests on a periodic basis (diagnostic service data interpretation rate). Worst-case the last check for incoming request messages was prior to the network layer reception of the functionally addressed request message. This would mean that the request would be stored in a buffer and earliest be processed the next time the scheduler checks for an incoming request. When server #2 processes the request then it determines that it does not need to answer, because it does not support the requested information. As shown in the figure above this would be after the completion of the response message of server #1 and even after the completion of the next request message transmitted by the client.

g) The client would send the next request right after the completion of all expected response messages

h) The completion of the request message is indicated in the servers via N_USData.ind, but only processed by the fast server #1, because server #2 did not yet handle the last request.

i) The completion of the new request is indicated in the client via N_USData.con.

This scenario does not only apply to functionally addressed requests but also to physically addressed requests in case the client does not want to receive any response message (responseRequired = no).

In order to handle the described scenarios the minimum times $P3_{CAN\_Physical}$ and $P3_{CAN\_Functional}$ between the end of a physically or functionally addressed request message and the start of a new physically or functionally addressed request message are defined for the client.

— The value of $P3_{CAN\_Physical}$ will be identical to $P2_{CAN\_Server\_max}$ for the physically addressed server. The timing applies to any physically addressed request message in any diagnostic session (default and non-default session) and in case no response is required by the server.

  The $P3_{CAN\_Physical}$ timer is started in the client each time a physically addressed request message with no response required is successfully transmitted onto the bus, which is indicated via N_USData.con in the client. When the client wants to transmit a new physically addressed request message following a previous request that was completely handled[5)] then this is only allowed in case the $P3_{CAN\_Physical}$ timer is no longer active at the time the client wants to transmit the physically addressed request message. In case $P3_{CAN\_Physical}$ would still be active at the point in time the client would like to transmit a new physically addressed request message then the transmission has to be postponed until $P3_{CAN\_Physical}$ is timed out.

5) Completely handled means that either no response is received in case no response is required, all expected responses to a functionally addressed request are received in case the responding servers are known and responses are required or a $P2_{CAN\_Client}$ timeout occurred in case the responding servers are not known and responses are required.

— The value of $P3_{CAN\_Functional}$ will be the maximum (worst-case) value of all functionally addressed server's $P2_{CAN\_Server\_max}$ for any functionally addressed request message in any diagnostic session (default and non-default session).

The $P3_{CAN\_Functional}$ timer is started in the client each time a functionally addressed request message with response required or with no response required is successfully transmitted onto the bus, which is indicated via N_USData.con in the client. When the client wants to transmit a new functionally addressed request message following a previous request that was completely handled[6)] then this is only allowed in case the $P3_{CAN\_Functional}$ timer is no longer active at the time the client wants to transmit the functionally addressed request message. In case $P3_{CAN\_Functional}$ would still be active at the point in time the client would like to transmit a new functionally addressed request message then the transmission has to be postponed until $P3_{CAN\_Functional}$ is timed out.

The requirement for the server is that it has to start with its response message within $P2_{CAN\_Server}$ (see section 5.3). This means that the diagnostic data interpretation rate of the server must be less than $P2_{CAN\_Server}$.

The following figure graphically depicts the $P3_{CAN\_Functional}$ timing handling for the client (based on the communication scenario given in the figure above). In addition the figure shows the handling of a functionally addressed TesterPresent (3E hex) request message in the client in case the $P3_{CAN\_Functional}$ timer is still active when $S3_{Client}$ times out (request will be postponed until $P3_{CAN\_Functional}$ times out). Following the figure a description can be found that references the points marked in the figure.

6) Completely handled means that either no response is received in case no response is required, all expected responses to a functionally addressed request are received in case the responding servers are known and responses are required or a $P2_{CAN\_Client}$ timeout occurred in case the responding servers are not known and responses are required.

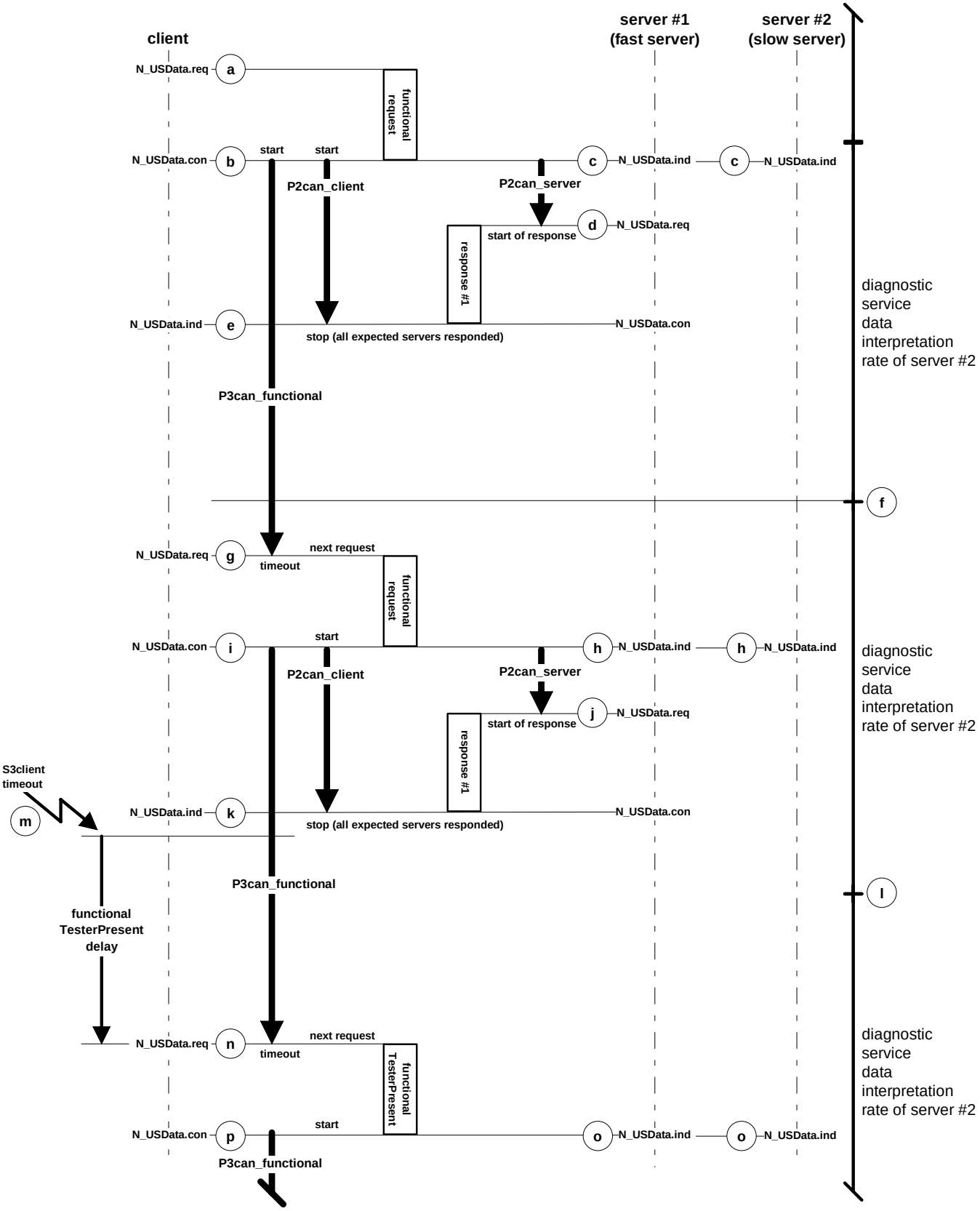

Figure 11 — Minimum time between functionally addressed request messages ($P3_{CAN\_Functional}$)

a) The diagnostic application of the client starts the transmission of a functionally addressed request message by issuing a N_USData.req to its network layer. The network layer transmits the request to the servers.

b) The completion of the request message is indicated in the client via N_USData.con. the client starts its $P2_{CAN\_Client}$ timer and furthermore its timer $P3_{CAN\_Functional}$.

c) The completion of the request message is indicated in the servers via N_USData.ind.

d) For the request message it is assumed that only server #1 supports the requested information, which means that there will be no response message from server #2. Server #1 is a fast server and can immediately process the received request message and transmits its response within $P2_{CAN\_Server}$.

e) Once the client receives the response message this is indicated via N_USData.ind. The client only expected a response message from server #1 therefore it stops its timer $P2_{CAN\_Client}$.

f) Server #2 is a slow server and interprets received requests on a periodic basis (diagnostic service data interpretation rate). Worst-case the last check for incoming request messages was right prior the network layer reception of the functionally addressed request message. This would mean that the request would be stored in a buffer and earliest be processed the next time the scheduler checks for an incoming request. When server #2 processes the request then it determines that it does not need to answer, because it does not support the requested information.

g) Even if the client has received all expected response messages to a functionally addressed request message it has to wait until $P3_{CAN\_Client}$ times out before it is allowed to transmit the next request message. At the point in time $P3_{CAN\_Client}$ times out the client transmits the next request message.

h) This new request is indicated in the servers via N_USData.ind and processed immediately by server #1 while server #2 processes the request the next time the scheduler checks for incoming request messages.

i) The completion of the new request is indicated in the client via N_USData.con and starts the $P3_{CAN\_Functional}$ timer in the client.

j) For the request message it is also assumed that only server #1 supports the requested information, which means that there will be no response message from server #2. Server #1 is a fast server and can immediately process the received request message and transmits its response within $P2_{CAN\_Server}$.

k) Once the client receives the response message this is indicated via N_USData.ind. The client only expected a response message from server #1 therefore it stops its timer $P2_{CAN\_Client}$.

l) Server #2 is a slow server and interprets received requests on a periodic basis (diagnostic service data interpretation rate). This would mean that the request would be stored in a buffer and earliest be processed the next time the scheduler checks for an incoming request. When server #2 processes the request then it determines that it does not need to answer, because it does not support the requested information.

m) The $S3_{Client}$ timer of the client times out, which forces the client to transmit a functionally addressed TesterPresent (3E hex) request message, not requiring a response message from the addressed server(s). Based on the situation that the $P3_{CAN\_Functional}$ timer is still active at this point in time the transmission of the TesterPresent (3E hex) has to be postponed until the expiration of the timer $P3_{CAN\_Functional}$.

n) When the $P3_{CAN\_Functional}$ timer times out the functionally addressed TesterPresent (3E hex) request can be transmitted by the client via N_USData.req.

o) The reception of the TesterPresent (3E hex) request message is indicated in the servers via N_USData.ind.

p) The completion of the TesterPresent (3E hex) request is indicated in the client via N_USData.con and starts the $P3_{CAN\_Functional}$ timer in the client.

The following figure graphically depicts the $P3_{CAN\_Physical}$ timing handling for the client. The figure shows the handling of a physically addressed request that does not require a response and it shows the handling of the functionally addressed TesterPresent (3E hex) request message in the client when $S3_{Client}$ times out. Following the figure a description can be found that references the points marked in the figure.

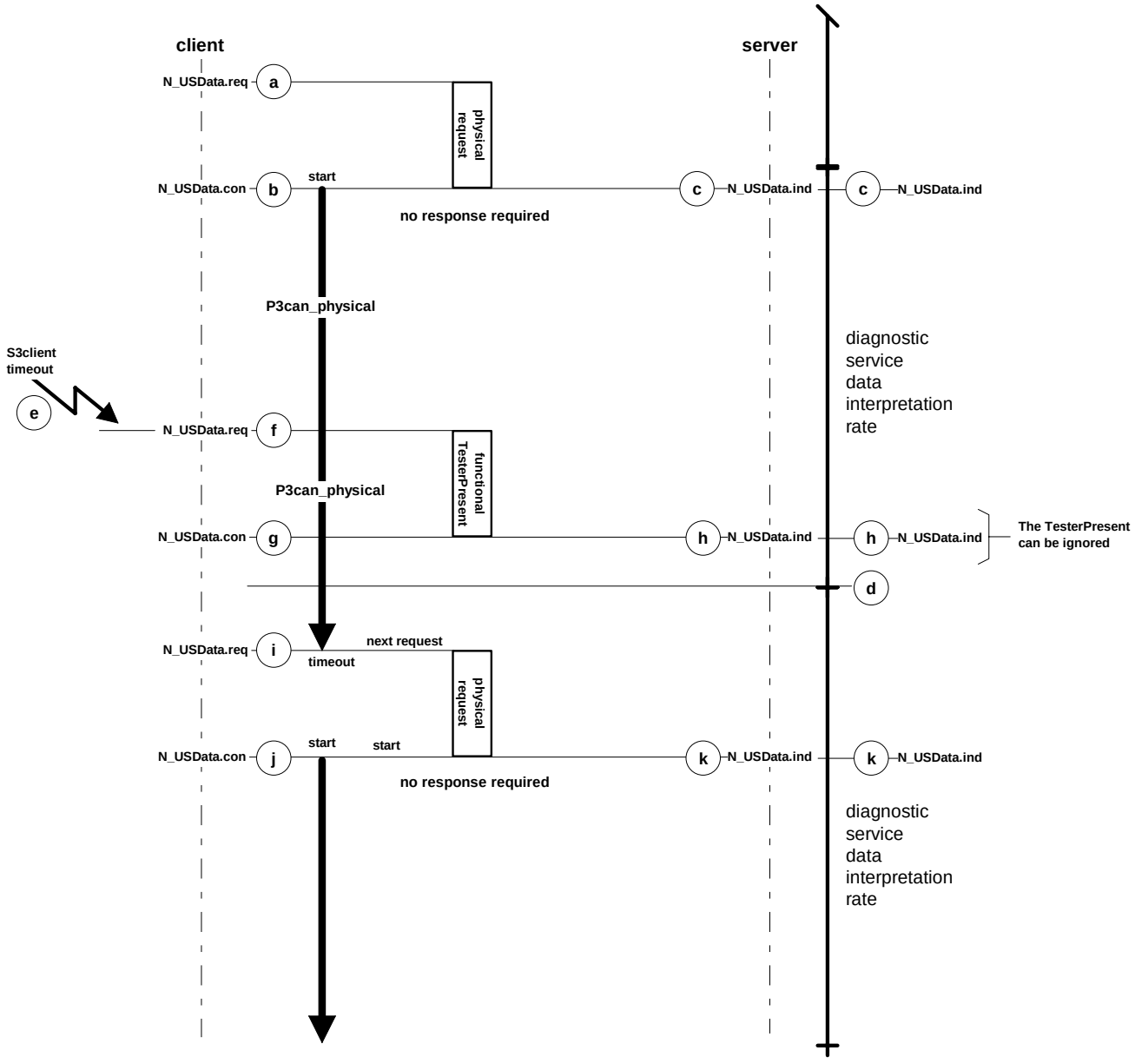

Figure 12 — Minimum time between physically addressed request messages ($P3_{CAN\_Physical}$)

a) The diagnostic application of the client starts the transmission of a physically addressed request message by issuing a N_USData.req to its network layer. The network layer transmits the request to the server.

b) The completion of the request message is indicated in the client via N_USData.con. The client now starts its $P3_{CAN\_Physical}$ timer. There is no response required to be transmitted therefore the client does not need to start its $P2_{CAN\_Client}$ timer.

c) The completion of the request message is indicated in the servers via N_USData.ind. In any non-default session the $S3_{Server}$ timer is now stopped.

d) The server interprets received requests on a periodic basis (diagnostic service data interpretation rate). The request is processed the next time the scheduler checks for incoming requests. The completed execution of the service would restart the $S3_{Server}$ timer during any non-default session.

e) The $S3_{Client}$ timer of the client times out, which forces the client to transmit a functionally addressed TesterPresent (3E hex) request message, not requiring a response message from the addressed server(s).

f) It is assumed that the $P3_{CAN\_Functional}$ timer is no active at this point in time, which means that the request is transmitted immediately.

g) The completion of the TesterPresent (3E hex) request message is indicated via N_USData.con in the client.

h) The reception of the TesterPresent (3E hex) request message is indicated in the servers via N_USData.ind. At this point in time the previous received physical request is still pending in the server (not yet processed) and the S3Server timer is stopped. Therefore the received TesterPresent (3E hex) request message can be ignored by the server.

i) When the $P3_{CAN\_Physical}$ timer times out in the client then the client can transmit the next physically addressed request message by issuing N_USData.req to its network layer.

j) The completion of the physically addressed request message is indicated in the client via N_USData.con. The client now starts its $P3_{CAN\_Physical}$ timer again. There is no response required to be transmitted therefore the client does not need to start its $P2_{CAN\_Client}$ timer.

k) The completion of the request message is indicated in the servers via N_USData.ind. In any non-default session the $S3_{Server}$ timer is now stopped.

5.3.5.4 Unsolicited response messages

Unsolicited messages are messages that are transmitted by the server(s) based on either a periodic scheduler (see service ReadDataByPeriodicIdentifier in section 8.2.4) or a configured trigger such as a change of a DTC status or a recordDataIdentifier value change (see service ResponseOnEvent in section 8.1.8).

Any unsolicited transmitted response message shall not reset the $S3_{Server}$ timer in the server. This avoids a diagnostic session keep-alive latch-up effect in the server for cases where a periodic message transmission is active or a timer-triggered event is configured in the server where the time interval between the events is smaller than $S3_{Server}$. The $S3_{Server}$ timer shall only be reset if the transmitted response message is the direct result of processing a request message and transmitting the final response message (such as the initial positive response that indicates that a request to schedule one or more periodicRecordDataIdentifiers is performed successfully). The requirements for the transmission of unsolicited response message can be found in the service descriptions of the services ReadDataByPeriodicIdentifier (see section 8.2.4) and ResponseOnEvent (section 8.1.8).

5.3.6 Error handling

The following tables define the error handling for the application layer and session management to be fulfilled by the client and the server during physical and functional communication. For all definitions it is assumed that the client and the server implement the application and session layer timing as defined within this standard.

Table 6 — Client error handling

Communication phase	Client error type	Client handling	
		Physical communication	**Functional communication**
Request transmission	N_USData.con from network layer with a negative result value.	The client shall repeat the last request, each after the time $P3_{CAN\_Physical}$ following the error indication. Restart $S3_{Client}$ in case of a physically addressed and sequentially transmitted TesterPresent, (because $S3_{Client}$ has been stopped based on the request message transmission).	The client shall repeat the last request, each after the time $P3_{CAN\_Functional}$ following the error indication.
$P2_{CAN\_Client}$ $P2^*_{CAN\_Client}$	Timeout	The client shall repeat the last request. Restart $S3_{Client}$ in case of a physically addressed and sequentially transmitted TesterPresent, (because $S3_{Client}$ has been stopped based on the request message transmission).	In case the client does not know the number of servers responding then this is the indication for the client that no further response messages are expected. No retry of the request message is required. The client has to completely receive all response messages that are in progress until it can continue with further requests.
			In case the client knows the number of responding servers then this is the indication for the client that not all expected servers responded. The client shall repeat the request after it has completely received any response message that is in progress at the point in time the timeout occurs.
Response reception	N_USData.ind from network layer with a negative result value.	The client shall repeat the last request. Restart $S3_{Client}$ in case of a physically addressed and sequentially transmitted TesterPresent, (because $S3_{Client}$ has been stopped based on the request message transmission).	The client shall repeat the last request after it has completely received any response message that is in progress at the point in time the error has been indicated.

The client error handling defined in the table above (client error handling) shall be performed for a maximum of two (2) times, which means that the worst-case of service request transmissions are three (3).

Table 7 — Server error handling

Communication phase	Server error type	Server handling
Request reception	N_USData.ind from network layer with a negative result value.	Restart $S3_{Server}$ timer (because it has been stopped based on the previously received FirstFrame indication). The server shall ignore the request.
$P2_{CAN\_Server}$ $P2_{CAN\_Client}$ $P2^*_{CAN\_Client}$	Timeout	N/A
Response transmission	N_USData.con from network layer with a negative result value.	Restart $S3_{Server}$ timer (because it has been stopped based on the previously received request message). The server shall not perform a retransmission of the response message.

6 Network layer interface

This international standard makes use of the network layer services defined in ISO 15765-2 for the transmission and reception of diagnostic messages. This section defines the mapping of the Application layer protocol data units (A_PDU) onto the Network layer protocol data units (N_PDU).

NOTE The network layer services are used to perform the application layer and diagnostic session management timing (see section 5.3).

6.1 FlowControl N_PCI parameter definition

The client shall not use the values of F1 hex – F9 hex for the Stmin parameter. These Stmin parameter values shall be supported by the server(s) if requested by the vehicle manufacturer.

6.2 Mapping of A_PDU onto N_PDU for message transmission

The parameters of the application layer protocol data unit defined to request the transmission of a diagnostic service request/response are mapped as follows onto the parameters of the network layer protocol data unit for the transmission of a message in the client/server.

Table 8 — Mapping of ServiceName.request/ServiceName.response A_PDU onto N_USData.request N_PDU

A_PDU parameter (Application Protocol Data Unit)	Description	N_PDU parameter (Network Protocol Data Unit)	Description
A_SA	**A**pplication **S**ource **A**ddress	N_SA	**N**etwork **S**ource **A**ddress
A_TA	**A**pplication **T**arget **A**ddress	N_TA	**N**etwork **T**arget **A**ddress
A_Tatype	**A**pplication **T**arget **A**ddress type	N_Tatype	**N**etwork **T**arget **A**ddress type
A_RA	**A**pplication **R**emote **A**ddress	N_AE	**N**etwork **A**ddress **E**xtension
A_PCI.SI	**A**pplication **P**rotocol **C**ontrol **I**nformation **S**ervice **I**dentifier	N_Data[0]	**N**etwork **D**ata
A_Data[0] – A_Data[n]	**A**pplication **D**ata	N_Data[1] N_Data[n+1]	**N**etwork **D**ata

The network layer confirmation of the successful transmission of the message (N_USData.con) is forwarded to the application, because it is needed in the application for starting those actions, which shall be executed right after the transmission of the request/response message (e.g. ECUReset, BaudrateChange, etc.).

6.3 Mapping of N_PDU onto A_PDU for message reception

The parameters of the network layer protocol data unit defined for the reception of a message are mapped as follows onto the parameters of the application layer protocol data unit for the confirmation/indication of the reception of a diagnostic response/request.

Table 9 — Mapping of N_USData.ind N_PDU onto ServiceName.conf/ServiceName.ind A_PDU

N_PDU parameter (Network Protocol Data Unit)	Description	A_PDU parameter (Application Protocol Data Unit)	Description
N_SA	**N**etwork **S**ource **A**ddress	A_SA	**A**pplication **S**ource **A**ddress
N_TA	**N**etwork **T**arget **A**ddress	A_TA	**A**pplication **T**arget **A**ddress
N_TAtype	**N**etwork **T**arget **A**ddress type	A_TAtype	**A**pplication **T**arget **A**ddress type
N_AE	**N**etwork **A**ddress **E**xtension	A_RA	**A**pplication **R**emote **A**ddress
N_Data[0]	**N**etwork **D**ata	A_PCI.SI	**A**pplication **P**rotocol **C**ontrol **I**nformation **S**ervice **I**dentifier
N_Data[1] N_Data[n+1]	**N**etwork **D**ata	A_Data[0] - A_Data[n]	**A**pplication **D**ata

The network layer indication for the reception of a FirstFrame N_PDU (N_USDataFirstFrame.ind) is not forwarded to the application, because it is only used within the application layer to perform the application layer timing (see section 5.3). Therefore no mapping of the N_USDataFirstFrame.ind N_PDU onto an A_PDU is defined.

7 Standardised diagnostic CAN identifiers

7.1 Legislated 11 bit OBD CAN identifiers

The following table is an extract out of ISO 15765-4, section "Data Link Layer Interface" and specifies the 11 bit CAN identifiers for legislated OBD. Those can also be used for enhanced diagnostics (e.g. the functional request CAN Id can be used for the functionally addressed TesterPresent (3E hex) request message to keep a non-defaultSession active).

NOTE ISO 15765-4 allows for max. 8 OBD related servers (ECUs), therefore 11 bit CAN identifiers for max. 8 servers are defined.

Table 10 — 11 bit CAN identifiers as defined in ISO 15765-4

CAN identifier (hex)	Description
7DF	CAN identifier for functionally addressed request messages sent by the client.
7E0	Physical request CAN identifier from the client to server #1
7E8	Physical response CAN identifier from server #1 to the client
7E1	Physical request CAN identifier from the client to server #2
7E9	Physical response CAN identifier from server #2 to the client
7E2	Physical request CAN identifier from the client to server #3
7EA	Physical response CAN identifier from server #3 to the client
7E3	Physical request CAN identifier from the client to server #4
7EB	Physical response CAN identifier from server #4 to the client
7E4	Physical request CAN identifier from the client to server #5
7EC	Physical response CAN identifier from server #5 to the client
7E5	Physical request CAN identifier from the client to server #6
7ED	Physical response CAN identifier from server #6 to the client
7E6	Physical request CAN identifier from the client to server #7
7EE	Physical response CAN identifier from server #7 to the client
7E7	Physical request CAN identifier from the client to server #8
7EF	Physical response CAN identifier from server #8 to the client

7.2 Legislated 29 bit OBD CAN identifiers

The following tables are an extract out of ISO 15765-4, section "Data Link Layer Interface" and specify the 29 bit CAN identifiers for legislated OBD. The 29 bit CAN identifiers comply with the Normal fixed addressing format specified in ISO 15765-2 and can also be used for enhanced diagnostics.

Table 11 — Summary of 29 bit CAN identifier format - normal fixed addressing

CAN Id bit position:	28 24	23 16	15 8	7 0
functional CAN Id	18 hex	DB hex	Target Address (TA)	Source Address (SA)
physical CAN Id	18 hex	DA hex	Target Address (TA)	Source Address (SA)

Table 12 — 29 bit CAN identifiers as defined in ISO 15765-4

CAN identifier (hex)	Description
18 DB 33 F1	CAN identifier for functionally addressed request messages sent by the client.
18 DA xx F1	Physical request CAN identifier from the client to server #xx
18 DA F1 xx	Physical response CAN identifier from server #xx to the client

NOTE The CAN identifier values given in the tables above use the default value for the priority information as specified in ISO 15765-2.

7.3 Enhanced diagnostics 29 bit CAN identifiers

This section specifies a standardised addressing and routing concept for CAN using 29 bit identifiers. The concept makes use of the well-known and approved mechanisms of the Internet Protocol (IP). By this means standardised algorithms for addressing and routing can be used for all nodes in the whole network independently from their positioning in sub networks.

This addressing and routing concept provides the following features:

— maximum flexibility during the design process of network structures,

— full customisation of network and node address,

— CAN controller hardware filter feature optimisation possible by assigning appropriate network and node address,

— gateways need to know only network addresses of the connected sub-networks instead of all addresses of their sub-network members.

The following sections specify the technical details of the CAN identifier structure, the structure of addresses, and subnet masks. A detailed description of the algorithms used for routing and broadcasting is also included.

7.3.1 Structure of 29 bit CAN identifier

The 29 bit CAN identifier structure specified in this document is compatible in regard to coexistence with the definitions in ISO 15765-2, -3, -4 standards and SAE J1939 recommended practice on the same physical CAN data link. Therefore the reserved bit in the 29 bit CAN identifier structure defined in SAE J1939-21 shall be used to determine whether a CAN identifier and frame is of SAE J1939 or ISO 15765 format. This enables the vehicle network designer to define non diagnostic messages and associated CAN identifiers customised according to his needs or utilise and benefit from the definitions in SAE J1939 in combination with a diagnostic services implementation as defined in ISO 15765-2,- 3, and -4.

For information about the structure of the SAE J1939 29 bit CAN identifier format refer to Table 13 — SAE J1939 structure of 29 bit CAN identifiers.

Table 13 — SAE J1939 structure of 29 bit CAN identifiers

29 bit CAN identifier							
28	**27**	**26**	**25**	**24**	**23 16**	**15 8**	**7 0**
Priority			**Reserved**	Data Page	PDU Format	PDU Specific	Source Address
			0			destination or PDU format extension	unique source address

The Table 14 — ISO 15765 structure of 29 bit CAN identifiers shows the structure of ISO 15765 CAN identifier that can be distinguished from the SAE J1939 format through the “SAE J1939 Reserved/ISO 15765 Format Type” bit 25 set to “1”. Thus, ISO 15765 formatted and SAE J1939 formatted 29 bit CAN identifiers can coexist on the same physical CAN bus system without interference.

Table 14 — ISO 15765 structure of 29 bit CAN identifiers

29 bit CAN identifier								
28	**27**	**26**	**25**	**24**	**23**	**22**	**21 11**	**10 0**
Priority			**Format Type**	TOS			Source Address	Destination Address
			1	Type of service			Unique source address	unique destination address

7.3.1.1 Priority field

The priority field is defined as specified in SAE J1939, to make use of the arbitration mechanism of CAN. Because the CAN identifier cannot be assigned freely anymore (source and target address are included in CAN identifier) the priority of a CAN message would be assigned by the sender (source address) and the receiver (target address) of that message indirectly. Eight (8) different priority levels are possible.

Priority level 6 (110b) shall be assigned to diagnostic request messages/frames.

7.3.1.2 Format type field

The format type bit shall be set to “1” to indicate that the ISO 15765 format of the 29 bit CAN identifier shall be used.

7.3.1.3 Type of service (TOS) field

The type of service field is used to be able to address different services of a node without having to assign different addresses to it. Thus, eight (8) different service types of a node can be addressed concurrently using a single destination address. The different types of services and their usage are defined in Table 15 — Definition of Types Of Service.

Table 15 — Definition of Types Of Service (TOS)

Value	Type Of Service (TOS)	Description
0x0 - 0x5	Non ISO 15765-3 messages	This range is reserved for information not defined in this document.
0x6	Network control message protocol / network management	The frames contain data, sent and received by gateways to supply information about the current state of subnets (e.g. network unreachable, network overload) and nodes (host unreachable).
0x7	ISO 15765 diagnostics	The diagnostic service of a node is addressed. The user data bytes of the CAN frame contain diagnostic requests (ISO 15765-3) using the transport layer (ISO 15765-2).

7.3.1.4 Source address

The source address contains the address of the sending entity. This information ensures the correct arbitration and can be used by the receiver of a message to address its replies. The structure of the source address is described in section 7.3.2 Structure of address.

7.3.1.5 Destination address

The destination address contains the address of the receiving entity. This can be a single node, the broadcast address of a network or a generic broadcast. The destination address is used by gateways to determine whether the CAN frame has to be routed to another CAN bus or not. The structure of the target address is described in section 7.3.2 Structure of address.

7.3.2 Structure of address

The source and destination address is encoded in the 29 bit CAN identifier with a length of 11 bits each. In the following sections the letters “X” and “Y” are used to represent a variable parameter.

7.3.2.1 Definition of address

An address consists of two parts:

a) **Network address:** The network address part consists of the first “X” sequential bits of the address and determines a node's network. The same network address shall be assigned to the nodes on one physical bus. The network address part must not have all bits set one (1). Thus, the minimum length for the network address part is two (2) bits. The maximum length is nine (9) bits because at least two (2) bits are needed to provide valid node address parts. The maximum number of possible subnets can be calculated as follows:

2^X - 1 (where “X” is the number of bits used for the network address part)

b) **Node address:** The node address part consists of the remaining “Y” (Y = 11 – X) sequential bits of the address and determines the node within a subnet. It must be unique within the subnet. All bits set to zero (0) and all bits set to one (1) are not allowed. Thus, the minimum length of the node address part is two (2) bits. The maximum length is nine (9) bits because at least two (2) bits are needed for the network address part. The maximum number of nodes per sub-network can be calculated as follows:

2^Y - 2 (where “Y” is the number of bits used for the node address part)

A node is assigned a unique address that has to be stored in the node’s internal memory. A node must receive messages with the node’s assigned address in the destination address field.

Table 16 — Example for source and destination address shows an example for source and destination address. The sending and the receiving nodes are on different sub-networks.

Table 16 — Example for source and destination address

29 bit CAN identifier																												
28	**27**	**26**	**25**	**24**	**23**	**22**	**21**										**11**	**10**										**0**
Priority 0x6			ISO 15765 format	TOS 0x7			Source Address 0x2ED											Destination Address 0x32F										
1	1	0	1	1	1	1	0	1	0	1	1	1	0	1	1	0	1	0	1	1	0	0	1	0	1	1	1	1

7.3.2.2 Subnet mask

The subnet mask assigns the number of bits used for the network address part and for the node address part.

The length of the subnet mask is 11 bits (same as the length of the address). The value of a subnet mask is assigned by setting the first “X” sequential bits set to one (1). The number of sequential bits set to one (1) select the network address part from the whole address. The remaining sequential bits set to zero (0) select the node address part from the whole address (see Table 17 — Example for sender’s subnet mask and Table 18 — Example for receiver’s subnet mask for examples of subnet masks for sender and receiver).

Due to the fixed length of a subnet mask and the first “X” sequential bits set to one (1), only the number of bits set to one (1) needs to be stored instead of the whole bit mask. Thus, a short notation is used to define a subnet mask.

Table 17 — Example for sender’s subnet mask

subnet mask										
10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0x7C0 (short notation: /5)										
network address part					**node address part**					
1	1	1	1	1	0	0	0	0	0	0

Table 18 — Example for receiver’s subnet mask

subnet mask										
10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0x7E0 (short notation: /6)										
network address part						**node address part**				
1	1	1	1	1	1	0	0	0	0	0

Each node is assigned a subnet mask that has to be stored in its internal memory. Nodes of the same subnet are assigned the same subnet mask.

7.3.2.3 Network address

The network address of a node can now be calculated using its assigned address and subnet mask. Therefore a simple bit by bit AND operation of address and subnet mask must be performed. See Table 19 — Example for sender’s network address and Table 20 — Example for receiver’s network address for an example of determining the network address of sender and receiver.

Table 19 — Example for sender’s network address

source address											
bit	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Address: 0x2ED	0	1	0	1	1	1	0	1	1	0	1
subnet mask: /5	1	1	1	1	1	0	0	0	0	0	0
network address: 0x2C0	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**

Table 20 — Example for receiver’s network address

destination address											
bit	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
address 0x32F	0	1	1	0	0	1	0	1	1	1	1
subnet mask: /6	1	1	1	1	1	1	0	0	0	0	0
network address: 0x320	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**

To describe a subnet, its network address and subnet mask are noted in the following form:

<network address> / <short subnet mask notation>

For the given examples this results in:

— sender's subnet: **0x2C0 / 5**

— receiver's subnet: **0x320 / 6**

This information is used by gateways for routing.

7.3.2.4 Broadcast address

The addressing allows for broadcasting messages to a single subnet or the whole network.

7.3.2.4.1 Generic broadcast (0x7FF)

To send a broadcast to the whole network the target address 0x7FF (all bits set to one (1)) shall be used. A message with that target address will be routed by all gateways. All nodes on the network must receive and process messages with destination address 0x7FF.

7.3.2.4.2 Subnet broadcast

To send a broadcast to a specific subnet the broadcast address of that subnet must be calculated. This is done by taking the destinations subnet information (network address and subnet mask) and setting all node address part bits (marked with zero (0) in subnet mask) to one (1). See Table 21 — Example for subnet broadcast to receiver's network for a subnet broadcast example for the receiver's subnet.

Table 21 — Example for subnet broadcast to receiver's network

destination address											
bit	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
network address: 0x320	0	1	1	0	0	1	0	0	0	0	0
subnet mask: /6	1	1	1	1	1	1	0	0	0	0	0
broadcast address: 0x33F	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**

Subnet broadcast messages are normally routed by gateways.

All nodes have to receive messages with the network address part equal to their own network address part and all bits set to "1" in the node address part of the destination address field.

7.3.3 Message retrieval

Each node on a subnet compares the destination address of a CAN frame with its own address. If those match the information contained is transferred to the next higher layer in the O.S.I. model for further processing.

7.3.4 Routing

Routing applies whenever nodes from physically disconnected subnets communicate with each other and their CAN frames have to be transferred from one subnet to another subnet. This is performed by additional nodes, which are physically connected to the network where the CAN frame is received and the network where the CAN frame has to be transmitted to, to reach its destination. Thus, a CAN frame may pass several gateways from its source subnet to its destination.

7.3.4.1 Network and subnet structure

Generally networks can be designed as needed when the following restrictions are respected:

— addresses must be unique,

— all nodes in a subnet have the same subnet mask,

— all nodes in a subnet have the same network address,

— whenever a network address was assigned to a subnet, no further network addresses in that address scope could be assigned to other networks because this would result in a routing problem,

— See Figure 13 — Network configuration. In this configuration four (4) subnets are connected. Three (3) subnets are connected through one gateway and the 4th subnet is connected through an additional gateway.

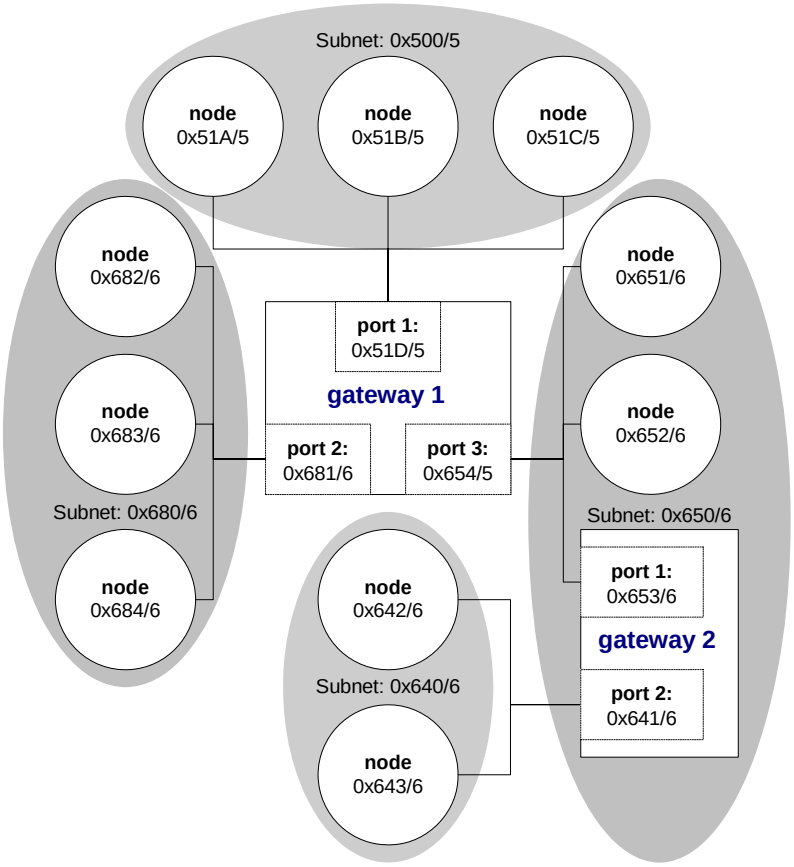

Figure 13 — Network configuration example

7.3.4.2 Gateways and routing

Gateways are nodes connected to more than one subnet and therefore able to transfer CAN frames from one subnet to another.

7.3.4.2.1 Ports

A port is the connection of a gateway to one physical subnet. A gateway must have at least two (2) ports. Each port is assigned a network address and subnet mask of the subnet it is connected to.

In Figure 13 — Network configuration example, which includes two (2) gateways, where gateway 1 has three ports and gateway 2 has two ports.

7.3.4.2.2 Routing table

To determine whether a CAN frame needs to be routed a routing table must be generated and stored in the gateway's memory. A routing entry contains the network address, subnet mask and the port on which the subnet can be reached. Such an entry must exist for each subnet that is connected (directly or indirectly) through this gateway.

See Table 22 — Routing table example for the network shown in Figure 13 — Network configuration example. Through hierarchical design of the networks 640/6 and 650/6 the routing table entries can be reduced to one entry 640/5.

Table 22 — Routing table example

subnet (network address/subnet mask)	port
Gateway 1	
500/5	1
680/6	2
640/5	3
Gateway 2	
500/5	1
680/6	1
650/6	1
640/6	2

7.3.4.2.3 Routing algorithm

A gateway receives all messages from the ports that are connected to the different subnets.

If the gateway is an addressable node then only one address out of the address scopes of the subnets connected directly to the ports of that gateway must be assigned. An additional message retrieval check is performed before the proper routing algorithm. If destination address is 0x7FF the message is copied to all ports except the one, on which the message was received. The normal routing algorithm is skipped.

After having received the message (A) the routing steps shown in Figure 14 — Routing algorithm sequence chart apply. The particular steps are described below.

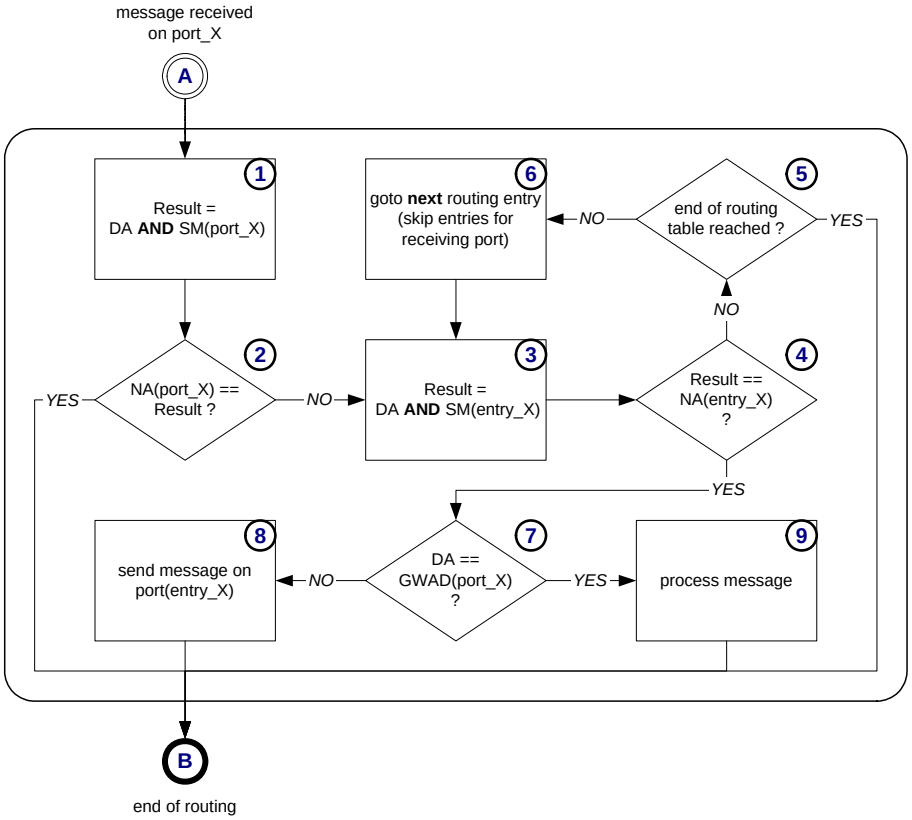

Figure 14 — Routing algorithm sequence chart

Legend:

DA	= destination address
GWAD	= gateway's address on port_X
NA	= network address
SM	= subnet mask
entry_X	= entry #X in the gateway's routing table
port_X	= port #X of the gateway

(A) A message is received on port “X”.

(1) A bit by bit logical AND operation is performed with destination address of the received message and subnet mask of the port on which the message was received

(2) The result is compared with the network address of the port on which the message was received. The network address of the port is either stored in the node’s memory or can be calculated using the address and subnet mask of that port. If the result and the network address are equal, the received message is a local message of the port’s subnet and no routing will apply (B). If the result and the port’s network address are different a routing analysis must be performed, continuing with step (3).

(3) A bit by bit logical AND operation is performed with destination address of the received message and the subnet mask of the current routing table entry.

(4) The result of the operation and the network address of the current routing table entry are compared. If those match the algorithm will continue with step (8), otherwise the algorithm will continue with step (5).

(5) If there are additional routing table entries the algorithm will continue with step (6). Otherwise no routing will apply (B).

(6) The next routing table entry is selected and the algorithm jumps back to step (3).

(7) The destination address of the message is compared with the gateway's address on the current port. This step is only needed if the gateway is an addressable node, otherwise the algorithm jumps directly to step (8). If the destination address is the address of the gateway for the current port the algorithm continues with step (9). If destination address and address of the gateway are not equal the algorithm is continued at step (8).

(8) The message is sent on the port of the routing table entry that matched the network address of the destination address.

(9) The message was addressed to gateway node and thus, it is processed by application.

(B) End of routing algorithm.

7.3.4.2.4 Routing example

In Figure 15 — Routing example from client 0x51A to server 0x642 a routing example is shown for a CAN message from the client with the address 0x51A to the server with the address 0x642 using the routing information from Table 22 — Routing table example.

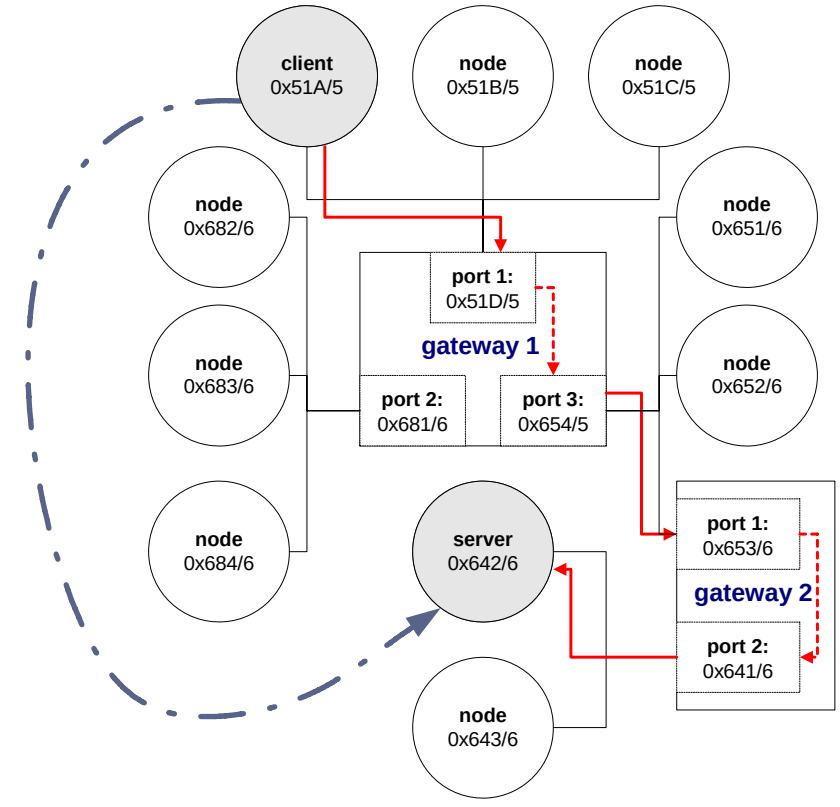

Figure 15 — Routing example from client 0x51A to server 0x642

Legend:

virtual connection
message path on CAN bus
message path in gateway

The following steps are performed by the gateways on reception of that message.

a) Gateway 1

— Analysis of CAN-ID: **DA = 0x642**

Table 23 — Gateway 1 routing decision

routing decision	network	port
(0x642 AND 0x7C0) = 0x640 != 500 → no local message → routing	500/5	1

Table 24 — Gateway 1 routing analysis

routing analysis	network	port
0x642 AND 0x7E0 = 0x640 != 680 → next entry	680/6	2
0x642 AND 0x7C0 = 0x640 = 640 → correct path	640/5	3

— check whether the message is addressed to gateway: 0x642 != 0x654

— forward message to port 3

b) Gateway 2

— Analysis of CAN-ID: DA = 0x642

Table 25 — Gateway 2 routing decision

routing decision	network	port
0x642 AND 0x7C0 = 0x640 != 650 → no local message → routing	650/6	1

Table 26 — Gateway 2 routing analysis

routing analysis	network	port
0x642 AND 0x7E0 = 0x640 = 640 → correct path	640/6	2

— check whether the message is addressed to gateway: 0x642 != 0x641

— forward message to port 2

8 Diagnostic services implementation

This section defines how the diagnostic services as defined in ISO 14229-1 apply onto CAN. For each applicable service the applicable sub-function and data parameters are defined.

NOTE The sub-function parameter definitions take into account that the most significant bit is used for the responseRequired parameter as defined in ISO 14229-1.

The purpose of the table below is to provide an overview of all diagnostic services as they are applicable for an implementation of Diagnostics on CAN. This table contains the sum of all applicable services. Certain applications using this international standard to implement Diagnostics on CAN may restrict the number of useable services and may categorise them in certain application areas / diagnostic sessions (e.g. default session, programming session, etc.). The table also defines for which service the responseRequired is allowed to be used when implementing the service onto CAN.

Table 27 — Diagnostics on CAN - Diagnostic services overview

Diagnostic service name (ISO 14229-1)	Service Id value (hex)	sub-function supported	responseRequired = 1 (no response) allowed[7)]	Section number
Diagnostic and Communication Management Functional Unit				
DiagnosticSessionControl	10	yes	yes	8.1.1
ECUReset	11	yes	yes	8.1.2
SecurityAccess	27	yes	-	8.1.3
CommunicationControl	28	yes	yes	8.1.4
TesterPresent	3E	yes	yes	8.1.5
SecuredDataTransmission	84	-	N/A	8.1.6
ControlDTCSetting	85	yes	yes	8.1.7
ResponseOnEvent	86	yes	yes	8.1.8
LinkControl	87	yes	yes	8.1.9
Data Transmission Functional Unit				
ReadDataByIdentifier	22	-	N/A	8.2.1
ReadMemoryByAddress	23	-	N/A	8.2.2
ReadScalingDataByIdentifier	24	-	N/A	8.2.3
ReadDataByPeriodicIdentifier	2A	-	N/A	8.2.4
DynamicallyDefineDataIdentifier	2C	yes	-	8.2.5
WriteDataByIdentifier	2E	-	N/A	8.2.6
WriteMemoryByAddress	3D	-	N/A	8.2.7
Stored Data Transmission Functional Unit				
ReadDTCInformation	19	yes	-	8.3.1
ClearDiagnosticInformation	14	-	N/A	8.3.2
Input/Output Control Functional Unit				
InputOutputControlByidentifier	2F	-	N/A	8.4.1

7) It is implied that responseRequired = 0 is supported by each service that utilises a sub-function parameter. It is the system designer's responsibility to assure that in case the client does not require a response message (responseRequired = 1) and the server might need more than $P2_{CAN\_Server}$ to process the request message that the client shall insert sufficient time between subsequent requests. There might be situations where a server is not able to perform the requested action nor being able to indicate the reason to the client.

Table 27 — Diagnostics on CAN - Diagnostic services overview

Diagnostic service name (ISO 14229-1)	Service Id value (hex)	sub-function supported	responseRequired = 1 (no response) allowed[7)]	Section number
Remote Activation Of Routine Functional Unit				
RoutineControl	31	yes	yes	8.5.1
Upload/Download Functional Unit				
RequestDownload	34	-	N/A	8.6.1
RequestUpload	35	-	N/A	8.6.2
TransferData	36	-	N/A	8.6.3
RequestTransferExit	37	-	N/A	8.6.4

8.1 Diagnostic and communication control functional unit

8.1.1 DiagnosticSessionControl (10 hex) service

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 28 — Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
01	**defaultSession**	U	DS
02	**ECUProgrammingSession**	U	ECUPS
03	**ECUExtendedDiagnosticSession**	U	ECUEDS

The tables below define the structure of the response message data parameter sessionParameterRecord as applicable for the implementation of this service on CAN.

Table 29 — sessionParameterRecord definition

Byte pos. in record	Description	Cvt	Hex Value	Mnemonic
	sessionParameterRecord[] #1 = [			SPREC_
#1	$P2_{Can\_Server\_max}$ (high byte)	M	00-FF	P2CSMH
#2	$P2_{Can\_Server\_max}$ (low byte)	M	00-FF	P2CSML
#3	$P2^*_{Can\_Server\_max}$ (high byte)	M	00-FF	P2ECSMH
#4	$P2^*_{Can\_Server\_max}$ (low byte)]	M	00-FF	P2ECSML

Table 30 — sessionParameterRecord content definition

Parameter	Description	# of bytes	Resolution	min value	max value
$P2_{Can\_Server\_max}$	Default $P2_{Can\_Server\_max}$ timing supported by the server for the activated diagnostic session.	2	1 ms	0 ms	65535 ms
$P2^*_{Can\_Server\_max}$	Enhanced (NRC 78 hex) $P2_{Can\_Server\_max}$ supported by the server for the activated diagnostic session.	2	10 ms	0 ms	655350 ms

8.1.2 ECUReset (11 hex) service

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 31 —Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
01	**hardReset**	U	HR
02	**keyOffOnReset**	U	KOFFONR
03	**softReset**	U	SR

8.1.3 SecurityAccess (27 hex) service

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 32 —Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
01	**requestSeed**	U	RSD
02	**sendKey**	U	SK
03, 05, 07-3F	**requestSeed**	U	RSD
04, 06, 08-40	**sendKey**	U	SK

8.1.4 CommunicationControl (28 hex) service

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 33 —Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**enableRxAndTx**	U	ERXTX
01	**enableRxAndDisableTx**	U	ERXDTX
02	**disableRxAndEnableTx**	U	DRXETX
03	**disableRxAndTx**	U	DRXTX

The table below defines the data parameters applicable for the implementation of this service on CAN.

Table 34 —Data parameter definition - communicationType

Bit 2-0	Description	Cvt	Mnemonic
001b	**application**	U	APPL
010b	**networkManagement**	U	NWM
100b	**diagnostic**	U	DIAG

8.1.5 TesterPresent (3E hex) service

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 35 —Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
00	**zeroSubFunction**	M	ZSUBF

8.1.6 SecuredDataTransmission (84 hex) service

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 36 —Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
01	**securedLinkSetup**	M	SLS
02	**securedModeDataTransmission**	M	SMDT
03	**closedSecuritySystemTransmission**	M	CSST

8.1.7 ControlDTCSetting (85 hex) service

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 37 —Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
01	**on**	M	ON
02	**off**	M	OFF

8.1.8 ResponseOnEvent (86 hex) service

The following requirements shall apply for this service when implemented on CAN:

a) Multiple ResponseOnEvent services may run concurrently with different requirements (different EventTypes, serviceToRespondTo-Records...) to start and stop diagnostic services.

b) While the ResponseOnEvent service is active, the server must be able to process concurrent diagnostic request and response messages accordingly. This should be accomplished with a (different) pair of serviceToRespondTo-request/response CAN identifiers. If the same diagnostic request/response CAN identifiers are used for diagnostic communication and the serviceToRespondTo-responses, the following restrictions shall apply:

   The server shall ignore an incoming diagnostic request after an event has occurred and the serviceToRespondTo-response is in progress, until the serviceToRespondTo-response is completed.

   1) After the client receives any response after sending a diagnostic request, the response must be classified according to the possible serviceToRespondTo-responses and the expected diagnostic responses that have been sent.

   2) If the response is a serviceToRespondTo-response (one of the possible responses set up with ResponseOnEvent-service), the client shall repeat the request after the serviceToRespondTo-response has been received completely.

   3) If the response is ambiguous (i.e. the response could originate from the serviceToRespondTo initiated by an event or from the response to a diagnostic request), the client shall present the response both as a serviceToRespondTo-response and as the response to the diagnostic request. The client shall not repeat the request with the exception of NegativeResponseCode busyRepeatRequest (21 hex). (Refer to the negative response code definitions in ISO 14229-1).

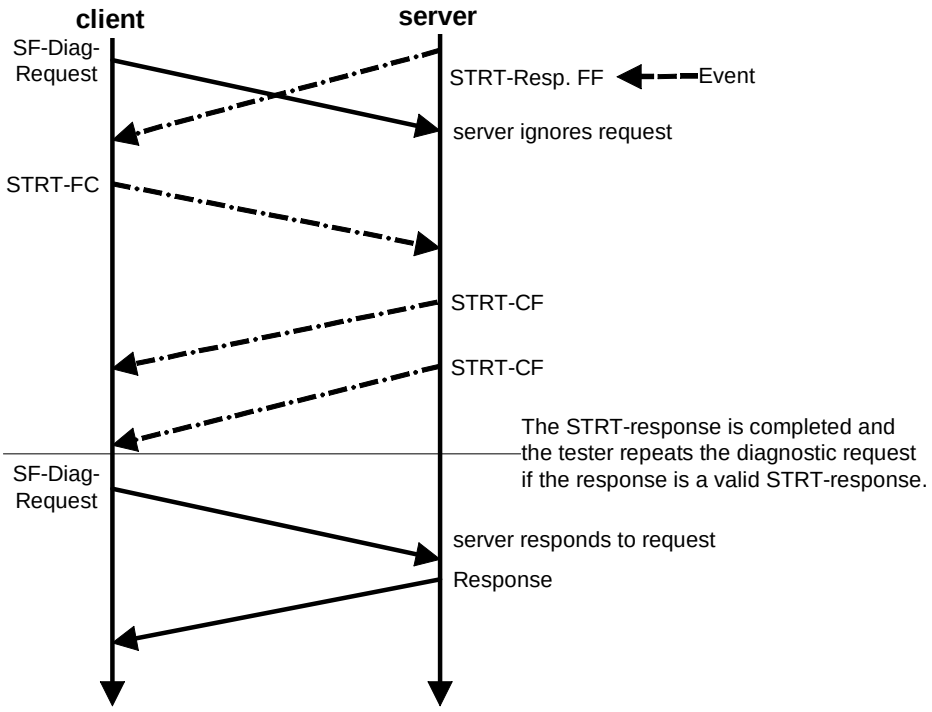

Figure 16 — Concurrent request when the event occurs

c) The ResponseOnEvent service shall only be allowed to use those diagnostic services available in the active diagnostic session.

d) While the ResponseOnEvent service is active, any change in a diagnostic session shall terminate the current ResponseOnEvent service(s). For instance, if a ResponseOnEvent service has been set up during extendedDiagnosticSession, it shall terminate when the server switches to the defaultSession.

e) If a ResponseOnEvent (86 hex) service has been set up during defaultSession then the following shall apply:

If Bit 6 of the eventType sub-function parameter is set to 0 (do not store event) then the event shall terminate when the server powers down The server shall not continue a ResponseOnEvent diagnostic service after a reset or power on (i.e. the ResponseOnEvent service is terminated).

If Bit 6 of the eventType sub-function parameter is set to 1 (store event), it shall resume sending serviceToRespondTo-responses according to the ResponseOnEvent-set up after a power cycle of the server.

f) The sub-function parameter value responseRequired = "no" should only be used for the eventType = stopResponseOnEvent, startResponseOnEvent or clearResponseOnEvent The server shall always return a response to the event-triggered response when the specified event is detected.

g) The server shall return a final positive response to indicate the ResponseOnEvent (86 hex) service has reached the end of the finite event window unless one of the following conditions apply:

— if eventTypes do not setup ResponseOnEvent, such as stopResponseOnEvent, startResponseOnEvent, clearResponseOnEvent or reportActivatedEvents

— if the infinite event window was established

  — if the Service has been deactivated before the event window was closed

  — Bit 6 of the eventType sub-function parameter is set to 0 (do not store) and the server powers down or resets

h) When the specified event is detected, the server shall respond immediately with the appropriate serviceToRespondTo-response message. The immediate serviceToRespondTo-response message shall not disrupt any other diagnostic request or response transmission already in progress (i.e. the serviceToRespondTo-response must be delayed until the current message transmission has been completed see figure below).

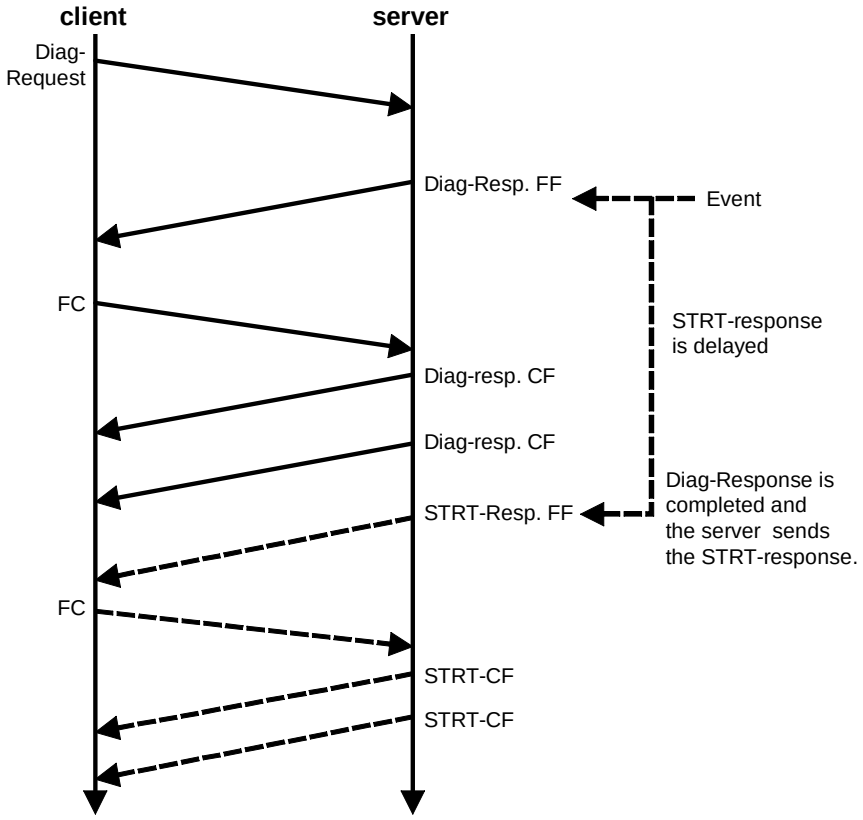

Figure 17 — Event occurrence during a message being in progress

i) The ResponseOnEvent service shall only apply to transient events and conditions. The server shall return a response once per event occurrence. For a condition that is continuously sustained over a period of time, the response service shall be executed only one time at the initial occurrence. In case the eventType is defined so that serviceToRespondTo-responses could occur at a high frequency then appropriate measures have to be taken in order to prevent back to back serviceToRespondTo-responses. A minimum separation time between serviceToRespondTo-responses could be part of the eventTypeRecord (vehicle manufacturer specific).

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 38 — eventType sub-function bit 6 definition - storageState

Bit 6 value	Description	Cvt	Mnemonic
0	**doNotStoreEvent**	M	DNSE
1	**storeEvent**	U	SE

Table 39 —Sub-function parameter definition

Hex (bit 5-0)	Description	Cvt	Mnemonic
00	**stopResponseOnEvent**	U	STPROE
01	**onDTCStatusChange**	U	ONDTCS
02	**onTimerInterrupt**	U	OTI
03	**onChangeOfRecordDataIdentifier**	U	OCOCID
04	**reportActivatedEvents**	U	RAE
05	**startResponseOnEvent**	U	STRTROE
06	**clearResponseOnEvent**	U	CLRROE

Table 40 —Data parameter definition - serviceToRespondToRecord.serviceId

Recommended services (ServiceToRespondTo)	RequestService Identifier (SId)
ReadDataByIdentifier	22 hex
ReadDTCInformation	19 hex
RoutineControl	31 hex
InputOutputControlByIdentifier	2F hex

8.1.9 LinkControl (87 hex) service

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 41 —Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
01	**verifyBaudrateTransitionWithFixedBaudrate**	U	VBTWFBR
02	**verifyBaudrateTransitionWithSpecificBaudrate**	U	VBTWSBR
03	**transitionBaudrate**	U	TB

8.2 Data transmission functional unit

8.2.1 ReadDataByIdentifier (22 hex) service

There are neither additional requirements nor restrictions defined for this service for its implementation on CAN.

8.2.2 ReadMemoryByAddress (23 hex) service

There are neither additional requirements nor restrictions defined for this service for its implementation on CAN.

8.2.3 ReadScalingDataByIdentifier(24 hex) service

There are neither additional requirements nor restrictions defined for this service for its implementation on CAN.

8.2.4 ReadDataByPeriodicIdentifier (2A hex) service

The two types of response messages as defined for this service in ISO 14229-1 are mapped onto CAN as follows:

— Response message type #1 (including the service identifier, the echo of the periodicRecordDataIdentifier and the data of the periodicRecordDataIdentifier):

  This type of response message is mapped onto a USDT[8)] message, using the same response CAN identifier as used for any other USDT response message. The USDT message for a single periodicRecordDataIdentifier shall not exceed the size of a single CAN frame which means that the complete USDT response message must fit into a SingleFrame N_PDU.

— Response message type #2 (including the periodicRecordDataIdentifier and the data of the periodicRecordDataIdentifier):

  This type of response message is mapped onto a UUDT[9)] message, using a different CAN identifier as used for the USDT response message. The UUDT message for a single periodicRecordDataIdentifier shall not exceed the size of a single CAN frame.

The mapping of the two response types lead to certain client and server requirements as listed in the tables given below.

Table 42 — Periodic transmission - requirements for the response type #1 message mapping

Message type	Client request requirements	Server response requirements	Further server restrictions
USDT uses the same CAN identifier for diagnostic communication and periodic transmission	no restrictions	Only single-frame responses for periodic transmission. Multi-frame responses to new (non-periodic-transmission) requests are possible.	Any other new incoming request shall be prioritised and the periodic transmission may be delayed.
			The periodic response is processed in the server as a regular USDT message (with protocol control information (PCI), service identifier (SId) and periodicRecordDataIdentifier) and is processed by the server network layer. This means that a maximum of 5 data bytes are available for the data of a periodicRecordDataIdentifier when using normal addressing and 4 data bytes when using extended addressing for the response message.
			For an incoming multi-frame request message any scheduled periodic transmission shall be delayed in the server immediately after the N_USDataFF.ind of a multi-frame request or the N_USData.ind of a SingleFrame request is processed by the application. Once the complete service is processed (including the final response message) the transmission of the periodic messages shall be continued.

8) USDT = Unacknowledged Segmented Data Transfer, ISO 15765-2 network layer, includes protocol control information for segmented data transmission

9) UUDT = Unacknowledged Unsegmented Data Transfer, single CAN frames, do not include protocol control information, which results in max. 7/8 data bytes for normal/extended addressing

Table 43 — Periodic transmission - requirements for response type #1 message mapping

Message type	Client request requirements	Server response requirements	Further server restrictions
UUDT uses a different CAN identifier for periodic transmission	no restrictions	Only single-frame responses for periodic transmission. Multi-frame responses to new (non-periodic-transmission) requests are possible.	The request for periodic transmission is processed as a regular diagnostic request and the response is sent via the network layer (as a USDT message with service identifier 6A hex).
			On receiving the N_USData.con that indicates the completion of the transmission of the positive response, the application starts an independent scheduler, which handles the periodic transmission.
			The scheduler in the server processes the periodic transmission as a single frame UUDT-message in a by-pass (i.e. writes the UUDT message directly to the CAN-controller/data link layer driver without using the network-layer).
			For an UUDT-message there is no need to include protocol control information (PCI) and service identifier (SId), only the periodic identifier is included, so a maximum of 7 data bytes can be used for the data of a periodicRecordDataIdentifier for normal addressing and 6 data bytes for extended addressing.

The following figures graphically depict the two types of periodic response messages, as the server should handle them. The figures furthermore show that the periodically transmitted response messages do not have any influence on the $S3_{Server}$ timer of the server. Following each figure a description can be found that references the points marked in the figure. For both figures it is assumed that a non-defaultSession has been activated prior to the configuration of the periodic scheduler (the ReadDataByPeriodicIdentifier service requires a non-defaultSession in order to be executed).

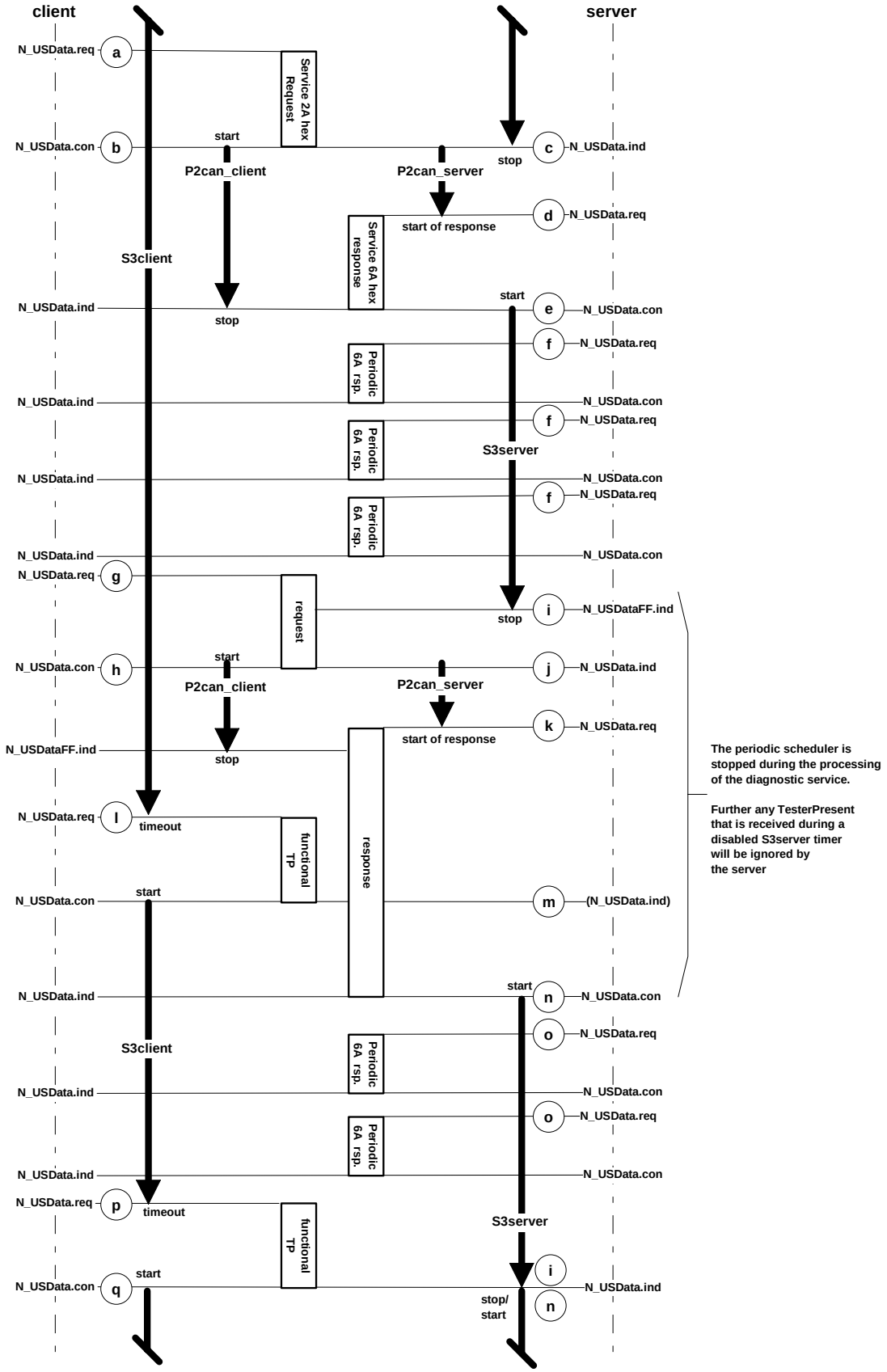

Figure 18: Response message type #1 handling

a) The diagnostic application of the client starts the transmission of the ReadDataByPeriodicIdentifier (2A hex) request message by issuing a N_USData.req to its network layer. The network layer transmits the ReadDataByPeriodicIdentifier (2A hex) request message to the server. The request message can either be a single frame message or a multi-frame message (depends on the number of periodicRecordDataIdentifier contained in the request message). For the example given it is assumed that the request message is a SingleFrame message.

b) The completion of the request message is indicated in the client via N_USData.con. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies.

c) The completion of the request message is indicated in the server via the N_USData.ind. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies. Furthermore the server stops its $S3_{Server}$ timer.

d) For the figure given it is assumed that the client requires a response from the server. The server has to transmit the ReadDataByPeriodicIdentifier positive response message to indicate that the request has been processed and that the transmission of the periodic messages will start afterwards.

e) The completion of the transmission of the ReadDataByPeriodicIdentifier response message is indicated in the server via N_USData.con. Now the server restarts its $S3_{Server}$ timer, which keeps the activated non-default session active as long as it does not time out.

f) The server starts to transmit the periodic response messages (SingleFrame message). Each periodic message utilizes the network layer protocol and uses the response CAN identifier that is also used for any other response message. Therefore the server issues a N_USData.req to the network layer each time a periodic message has to be transmitted and no other service is currently in the process of being handled by the server. For the example given it is assumed that the server is able to transmit three (3) periodic messages prior to the next request message that is issued by the client. The transmission of the periodic response messages has no influence on the $S3_{Server}$ timer (see section 5.3.5.4).

g) The diagnostic application of the client starts the transmission of the next request message by issuing a N_USData.req to its network layer. The network layer transmits the request message to the server. The request message can either be a single frame message or a multi-frame message. For the example given it is assumed that the request message is a multi-frame message.

h) The completion of the request message is indicated in the client via N_USData.con. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies.

i) Once the start of a request message is indicated in the server via N_USDataFF.ind (or N_USData.ind for SingleFrame request messages) while a periodic scheduler is active, the server must temporarily stop the periodic scheduler for the duration of processing the received request message. Furthermore any time the server is in the process of handling any diagnostic service it stops its $S3_{Server}$ timer.

j) The completion of the multi-frame request message is indicated in the server via the N_USData.ind. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies. The scheduler for the transmission of the periodic messages remains disabled.

k) For the figure given it is assumed that the client requires a response from the server. The server has to transmit the positive (or negative) response message via issuing N_USData.req to its network layer. For the example it is assumed that the response is a multi-frame message.

l) When the $S3_{Client}$ timer times out in the client then the client transmits a functionally addressed TesterPresent (3E hex) request message to reset the $S3_{Server}$ timer in the server.

m) The server is in the process of transmitting the multi-frame response of the previous request. Therefore the server must not act on the received TesterPresent (3E hex) request message, because its $S3_{Server}$ timer is not yet re-activated.

n) When the diagnostic service is completely processed then the server restarts its $S3_{Server}$ timer. This means that any diagnostic service, including TesterPresent (3E hex), resets the $S3_{Server}$ timer. A diagnostic service is meant to be in progress any time between the start of the reception of the request message (N_USDataFF.ind or N_USData.ind receive) and the completion of the transmission of the response message in case a response message is required or the completion of any action that is caused by the request in case no response message is required (point in time reached that would cause the start of the response message). This includes negative response messages including response code 78 hex. The server re-enables the periodic scheduler when the service is completely processed (final response message completely transmitted).

o) The server restarts the transmission of the periodic response messages (SingleFrame message). Each periodic message utilizes the network layer protocol and uses the response CAN identifier that is also used for any other response message. Therefore the server issues a N_USData.req to the network layer each time a periodic message has to be transmitted and no other service is currently in the process of being handled by the server. The transmission of the periodic response messages has no influence on the $S3_{Server}$ timer (see section 5.3.5.4).

p) Once the $S3_{Client}$ timer is started in the client (non-defaultSession active) this causes the transmission of a functionally addressed TesterPresent (3E hex) request message, which does not require a response message, each time the $S3_{Client}$ timer times out.

q) Upon the indication of the completed transmission of the TesterPresent (3E hex) request message via N_USData.con of its network layer the client once again starts its $S3_{Client}$ timer. This means that the functionally addressed TesterPresent (3E hex) request message is sent on a periodic basis every time $S3_{Client}$ times out.

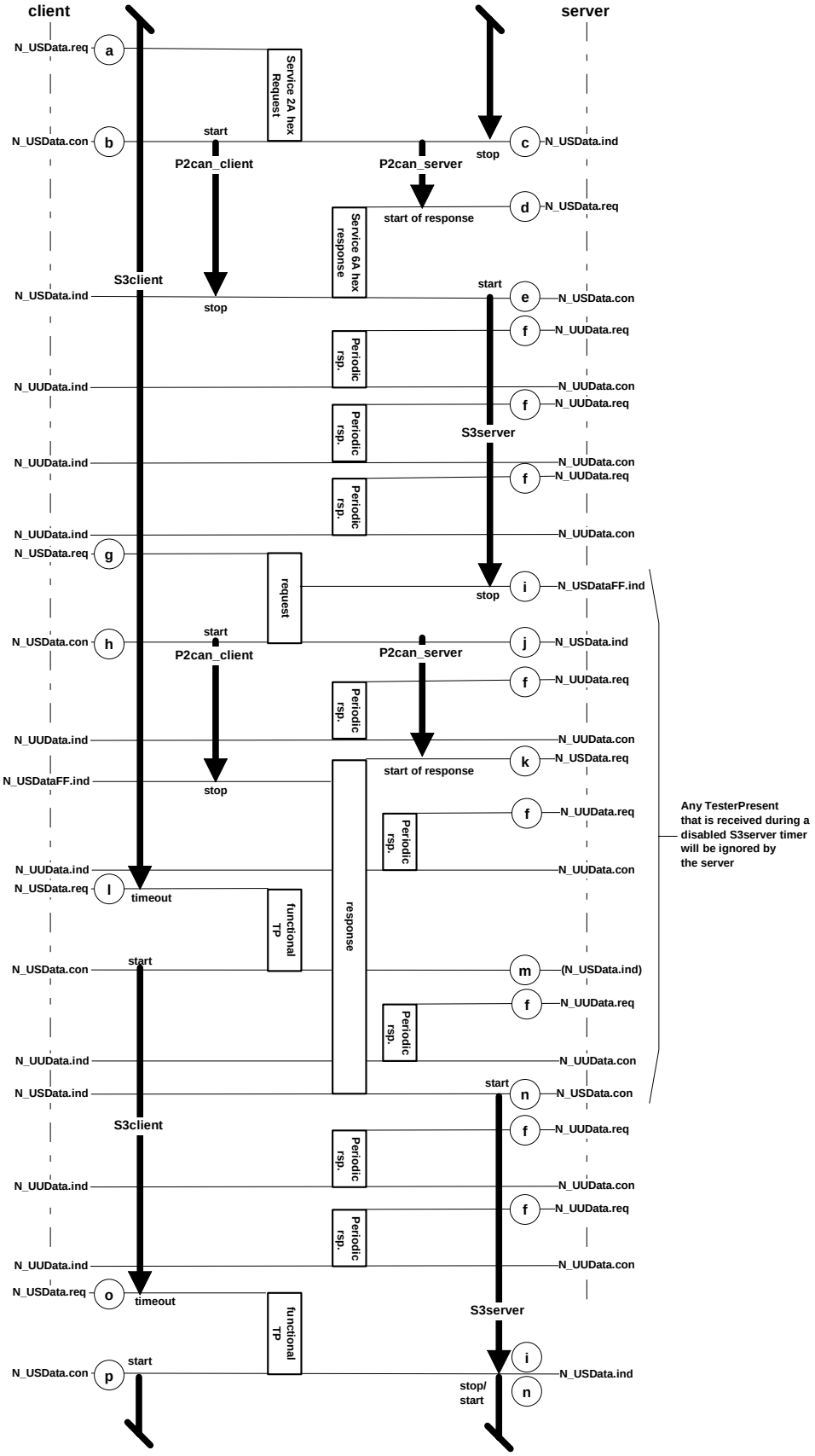

Figure 19: Response message type #2 handling

a) The diagnostic application of the client starts the transmission of the ReadDataByPeriodicIdentifier (2A hex) request message by issuing a N_USData.req to its network layer. The network layer transmits the ReadDataByPeriodicIdentifier (2A hex) request message to the server. The request message can either be a single frame message or a multi-frame message (depends on the number of periodicRecordDataIdentifier contained in the request message). For the example given it is assumed that the request message is a SingleFrame message.

b) The completion of the request message is indicated in the client via N_USData.con. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies.

c) The completion of the request message is indicated in the server via the N_USData.ind. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies. Furthermore the server stops its $S3_{Server}$ timer.

d) For the figure given it is assumed that the client requires a response from the server. The server has to transmit the ReadDataByPeriodicIdentifier positive response message to indicate that the request has been processed and that the transmission of the periodic messages will start afterwards.

e) The completion of the transmission of the ReadDataByPeriodicIdentifier response message is indicated in the server via N_USData.con. Now the server restarts its $S3_{Server}$ timer, which keeps the activated non-default session active as long as it does not time out.

f) The server starts to transmit the periodic response messages (single frame message). Each periodic message is a UUDT message and uses a different CAN identifier as used for any other response message (USDT CAN identifier). Therefore the server issues a N_UUData.req each time a periodic message has to be transmitted independent of any other service currently processed by the server. This means that the transmission of the periodic response messages continues even when the server is in the process of handling another diagnostic service request. The transmission of the periodic response messages has no influence on the $S3_{Server}$ timer (see section 5.3.5.4).

g) The diagnostic application of the client starts the transmission of the next request message by issuing a N_USData.req to its network layer. The network layer transmits the request message to the server. The request message can either be a single frame message or a multi-frame message. For the example given it is assumed that the request message is a multi-frame message.

h) The completion of the request message is indicated in the client via N_USData.con. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies.

i) The start of a request message is indicated in the server via N_USDataFF.ind (or N_USData.ind for SingleFrame request messages) while a periodic scheduler is active. The server does not stop the periodic scheduler for the duration of processing the received request message. This means that the server transmits further periodic messages for the duration of processing the diagnostic service. The client must be aware of receiving these periodic response messages. Furthermore any time the server is in the process of handling any diagnostic service it stops its $S3_{Server}$ timer.

j) The completion of the multi-frame request message is indicated in the server via the N_USData.ind. Now the response timing as described in section 5.3.5.1.1 and 5.3.5.1.2 applies.

k) For the figure given it is assumed that the client requires a response from the server. The server has to transmit the positive (or negative) response message via issuing N_USData.req to its network layer. For the example it is assumed that the response is a multi-frame message. While the multi-frame response message is transmitted by the network layer the periodic scheduler continues to transmit the periodic response messages.

l) When the $S3_{Client}$ timer times out in the client then the client transmits a functionally addressed TesterPresent (3E hex) request message to reset the $S3_{Server}$ timer in the server.

m) The server is in the process of transmitting the multi-frame response of the previous request. Therefore the server must not act on the received TesterPresent (3E hex) request message, because its $S3_{Server}$ timer is not yet re-activated.

n) When the diagnostic service is completely processed then the server restarts its $S3_{Server}$ timer. This means that any diagnostic service, including TesterPresent (3E hex), resets the $S3_{Server}$ timer. A diagnostic service is meant to be in progress any time between the start of the reception of the request message (N_USDataFF.ind or N_USData.ind receive) and the completion of the transmission of the response message in case a response message is required or the completion of any action that is caused by the request in case no response message is required (point in time reached that would cause the start of the response message). This includes negative response messages including response code 78 hex.

o) Once the $S3_{Client}$ timer is started in the client (non-defaultSession active) this causes the transmission of a functionally addressed TesterPresent (3E hex) request message, which does not require a response message, each time the $S3_{Client}$ timer times out.

p) Upon the indication of the completed transmission of the TesterPresent (3E hex) request message via N_USData.con of its network layer the client once again starts its $S3_{Client}$ timer. This means that the functionally addressed TesterPresent (3E hex) request message is sent on a periodic basis every time $S3_{Client}$ times out.

The table below defines the data parameters applicable for the implementation of this service on CAN.

Table 44 —Data parameter definition - transmissionMode

Hex	Description	Cvt	Mnemonic
01	**sendAtSlowRate**	U	SASR
02	**sendAtMediumRate**	U	SAMR
03	**sendAtFastRate**	U	SAFR
04	**stopSending**	U	SS

8.2.5 DynamicallyDefineDataIdentifier (2C hex) service

When the client dynamically defines a periodicRecordDataIdentifier and the total length of the dynamically defined periodicRecordDataIdentifier exceeds the maximum length that fits into a single frame periodic response message, then the request shall be rejected with a negative response message including negative response code 31 hex (requestOutOfRange). See ReadDataByPeriodicIdentifier (section 8.2.4) for details regarding the periodic response message format.

When multiple DynamicallyDefineDataIdentifier requests messages are used to configure a single periodicRecordDataIdentifier and the server detects the overrun of the maximum number of bytes during a subsequent request for this periodicRecordDataIdentifier then the server shall leave the definition of the periodicRecordDataIdentifier as it was prior to the request that lead to the overrun.

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 45 —Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
01	**defineByIdentifier**	U	DBID
02	**defineByMemoryAddress**	U	DBMA
03	**clearDynamicallyDefinedDataIdentifier**	U	CDDDI

8.2.6 WriteDataByIdentifier (2E hex) service

There are neither additional requirements nor restrictions defined for this service for its implementation on CAN.

8.2.7 WriteMemoryByAddress (3D hex) service

There are neither additional requirements nor restrictions defined for this service for its implementation on CAN.

8.3 Stored data transmission functional unit

8.3.1 ReadDTCInformation (19 hex) service

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 46 —Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
01	**reportNumberOfDTCByStatusMask**	U	RNODTCBSM
02	**reportDTCByStatusMask**	M	RDTCBSM
03	**reportDTCSnapshotIdentification**	U	RDTCSSI
04	**reportDTCSnapshotRecordByDTCNumber**	U	RDTCSSBDTC
05	**reportDTCSnapshotRecordByRecordNumber**	U	RDTCSSBRN
06	**reportDTCExtendedDataRecordByDTCNumber**	U	RDTCEDRBDN
07	**reportNumberOfDTCBySeverityMaskRecord**	U	RNODTCBSMR
08	**reportDTCBySeverityMaskRecord**	U	RDTCBSMR
09	**reportSeverityInformationOfDTC**	U	RSIODTC
0A	**reportSupportedDTC**	U	RDTCEDRBDN
0B	**reportFirstTestFailedDTC**	U	RFVDTC
0C	**reportFirstConfirmedDTC**	U	RFCDTC
0D	**reportMostRecentTestFailedDTC**	U	RMRVDTC
0E	**reportMostRecentConfirmedDTC**	U	RMRCDTC

The table below defines the DTC status bits applicable for the implementation of this service on CAN.

Table 47 —DTC status bit definitions

Bit	Description	Cvt		Mnemonic
		Emission	Non-Emission	
0	**testFailed**	U	U	TF
1	**testFailedThisMonitoringCycle**	M	C_1	TFTMC
2	**pendingDTC**	M	U	PDTC
3	**confirmedDTC**	M	M	CDTC
4	**testNotCompletedSinceLastClear**	C_2	C_2	TNCSLC
5	**testFailedSinceLastClear**	C_2	C_2	TFSLC
6	**testNotCompletedThisMonitoringCycle**	M	M	TNCTMC
7	**warningIndicatorRequested**	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.				

In case a DTCFailureTypeByte is used when implementing this service on CAN then the DTCFailureTypeByte definitions as defined in ISO 14229-1 apply.

8.3.2 ClearDiagnosticInformation (14 hex) service

The table below defines the data parameters applicable for the implementation of this service on CAN.

Table 48 —Data parameter definition - groupOfDTC

Hex	Description	Cvt	Mnemonic
000000 - FFFFFE	Individual / Single DTC	U	SDTC
FFFFFF	All Groups (all DTCs)	M	AG

8.4 Input/Output control functional unit

8.4.1 InputOutputControlByIdentifier (2F hex) service

In case the first byte of the controlOptionRecord is used as an InputOutputControlParameter then the table below defines the data parameters applicable for the implementation of this service on CAN.

Table 49 —Data parameter definition - inputOutputControlParameter

Hex	Description	Cvt	Mnemonic
00	**returnControlToECU**	U	RCTECU
01	**resetToDefault**	U	RTD
02	**freezeCurrentState**	U	FCS
03	**shortTermAdjustment**	U	STA

8.5 Remote activation of routine functional unit

8.5.1 RoutineControl (31 hex) service

The table below defines the sub-function parameters applicable for the implementation of this service on CAN.

Table 50 —Sub-function parameter definition

Hex (bit 6-0)	Description	Cvt	Mnemonic
01	**startRoutine**	U	STR
02	**stopRoutine**	U	STPR
03	**requestRoutineResults**	U	RRR

8.6 Upload/Download functional unit

8.6.1 RequestDownload (34 hex) service

There are neither additional requirements nor restrictions defined for this service for its implementation on CAN.

8.6.2 RequestUpload (35 hex) service

There are neither additional requirements nor restrictions defined for this service for its implementation on CAN.

8.6.3 TransferData (36 hex) service

There are neither additional requirements nor restrictions defined for this service for its implementation on CAN.

8.6.4 RequestTransferExit (37 hex) service

There are neither additional requirements nor restrictions defined for this service for its implementation on CAN.

9 Non-volatile server memory programming process

9.1 General information

This section defines a framework for the physically oriented download of one or multiple application software/data modules into non-volatile server memory. The defined non-volatile server memory programming sequence addresses:

— vehicle manufacturer specific needs in performing certain steps during the programming process, while being compliant with the general service execution requirements as specified in ISO 14229-1 (such as the sequential order of services and the session management),

— that the CAN bus is a network with multiple nodes connected, which interact with each other, using normal communication CAN messages,

— to either use a physically oriented vehicle approach (point-to-point communication - servers do not support functional diagnostic communication) or a functionally oriented vehicle approach (point-to-point and point-to-multiple communication - servers support functional diagnostic communication). A single vehicle shall only support one of the above mentioned vehicle approaches.

The programming sequence is divided into two programming phases. All steps are categorised based on the following types:

— Standardised steps - this type of step is mandatory. The client and the server shall behave as specified,

— Optional/recommended steps - this type of step is optional. Optional steps contain recommendations how an operation shall be performed. In case the specified functionality is used then the client and the server shall behave as specified,

— Vehicle manufacturer specific steps - the usage and content of this step is left to the discretion of the vehicle manufacturer and has to be in compliance with ISO 14229-1 and ISO 15765-3.

The defined steps can either be

— functionally addressed to all nodes on the CAN network (functionally oriented vehicle approach, servers support functional diagnostic communication), or

— physically addressed to each node on the CAN network (physically oriented vehicle approach, servers do not support functional diagnostic communication).

Each step of the two programming phases of the programming procedure will specify the allowed addressing method for that step. The vehicle manufacturer specific steps can either by functionally or physically addressed (depends on the OEM requirements).

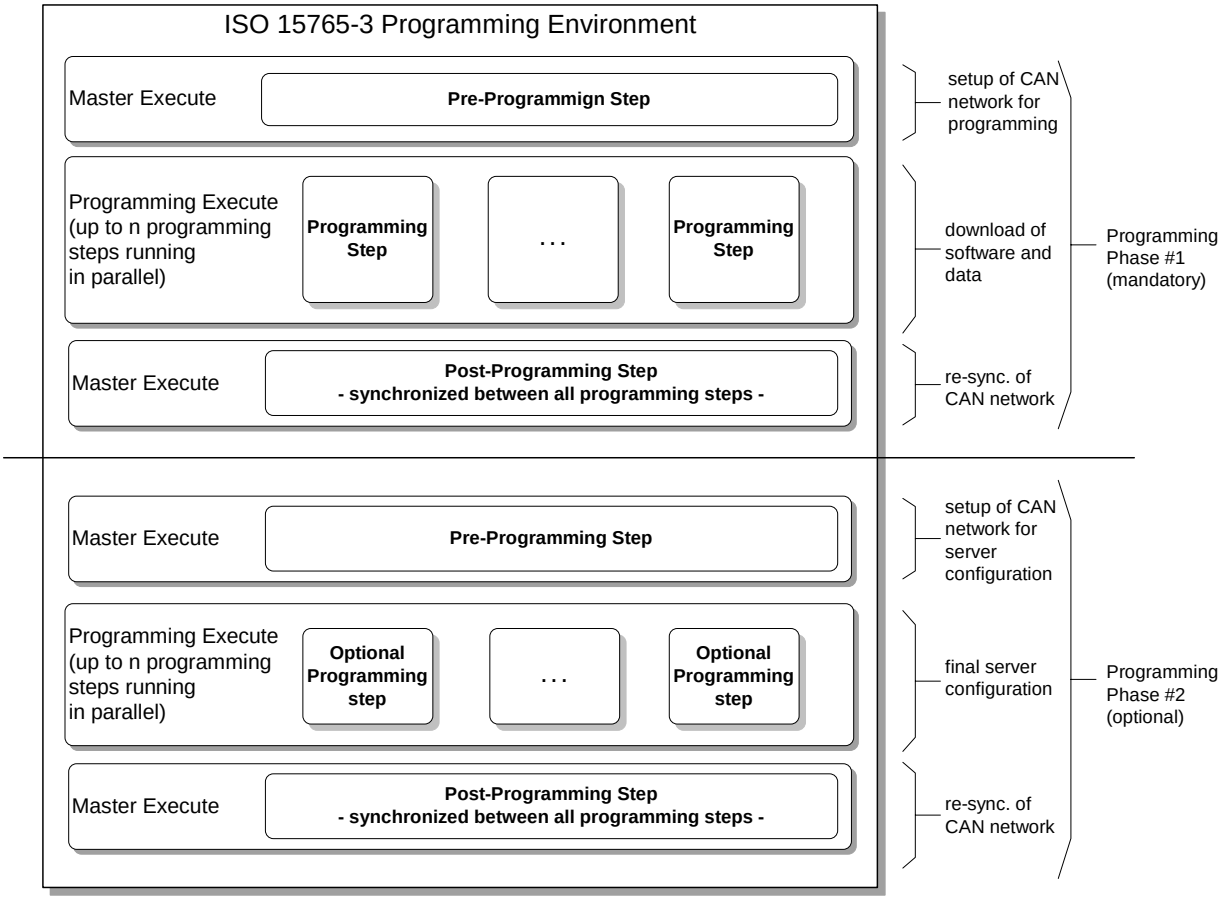

Figure 20 — ISO 15765-3 non-volatile server memory programming process overview

a) Programming phase #1 – download of application software and/or application data

Within programming phase #1 the application software/data is transferred to the server.

Optional Pre-Programming step – setup of CAN network for programming

The pre-programming step of phase #1 is optional to be used to prepare the CAN network for a programming event of one or multiple servers. This step provides certain hooks where a vehicle manufacturer can insert specific operations that are required for the OEM vehicle's CAN network (e.g. perform wake-up, determine communication parameters, read server identification data, etc.).

This step also contains provisions to increase the baudrate to improve download performance. The usage of this functionality is optional and can only be performed in case of a functionally oriented vehicle approach (functional diagnostic communication supported by the servers).

The request messages of this step can either be physically or functionally addressed.

Server Programming step – download of application software and application data

The server programming step of phase #1 is used to program one or multiple servers (download of application software and/or application data and/or boot software).

Within this step only physical addressing is used by the client, which allows for parallel or sequential programming of multiple nodes. In case the pre-programming step is not used then the DiagnosticSessionControl (10 hex) with sub-function programmingSession can also be performed using functional addressing.

At the end of this step the reset of the re-programmed server(s) is optional. The use of the reset leads to the requirement to implement programming phase #2 in order to finally conclude the programming event by physically clearing DTCs in the re-programmed server(s), because after the physical reset during this step the re-programmed server(s) enable(s) the default session and perform(s) their normal mode of operation while the remaining server(s) have still disabled normal communication. The re-programmed server(s) will potentially set DTCs.

Furthermore it has to be considered that the re-programmed server could activate a new set of diagnostic CAN identifiers, which differs to the ones used when performing a programming event (see section 9.3).

If either the server that was re-programmed does not change its communication parameters or the client knows the changed communication parameters then following the reset certain configuration data can be written to the re-programmed server.

Post-Programming step – re-synchronisation of CAN network after programming

The post-programming step of phase #1 concludes the programming phase #1. This step is performed when the programming step of each reprogrammed server is finished.

The request messages of this steps can either be physically or functionally addressed.

The CAN network is transitioned to its normal mode of operation. This can either be done via a reset using the ECUReset (11 hex) service or an explicit transition to the default session via the DiagnosticSessionControl (10 hex) service. The DiagnosticSessionControl (10 hex) service shall not enable a potentially present valid application software in the server (no implicit ECUReset).

b) Programming phase #2 – server configuration (optional)

Programming phase #2 is an optional phase where the client can perform further action that are needed to finally conclude a programming event (e.g. write the VIN, trigger Immobilizer learn-routine, etc.). For example if the server(s) that has (have) been re-programmed is (are) physically reset during the server programming step of programming phase #1 then DTCs must be cleared in this server(s).

When executing this phase the downloaded application software/application data is running/activated in the server and the server provides its full diagnostic functionality.

Pre-Programming step – setup of CAN network for server configuration

The pre-programming step of phase #2 is used to prepare the CAN network for the programming step of phase #2. This step is an optional step and provides certain hooks where a vehicle manufacturer can insert specific operations that are required for OEM vehicle's CAN network (e.g. wake-up, determine communication parameters, etc.).

The request messages of this steps can either be physically or functionally addressed.

Programming step – final server configuration

The programming step is used to e.g. write data (e.g. VIN) after the server reset.

The content of this step is vehicle manufacturer specific.

If the server(s) that has (have) been re-programmed are physically reset at the end of the server programming step of programming phase #1 then DTCs must be cleared in this server(s) during the programming step of phase #2.

The request messages of this steps are physically addressed.

Post-Programming step – re-synchronisation of CAN network after final server configuration

The post-programming step concludes programming phase #2. This step is performed when the programming step of each reprogrammed server is finished. The CAN network is transitioned to its normal mode of operation.

This step can either be functionally oriented (servers support functional diagnostic communication) or physically oriented (servers do not support functional diagnostic communication).

The request messages of this steps can either be physically or functionally addressed.

9.2 Detailed programming sequence

This section specifies detailed steps of the two (2) programming phases.

9.2.1 Programming phase #1 – download of application software and/or application data

9.2.1.1 Pre-Programming step of phase #1 – setup of CAN network for programming

The following figure graphically depicts the functionality embedded in the pre-programming step. Following the figure a textual description can be found, which references the points in the figure.

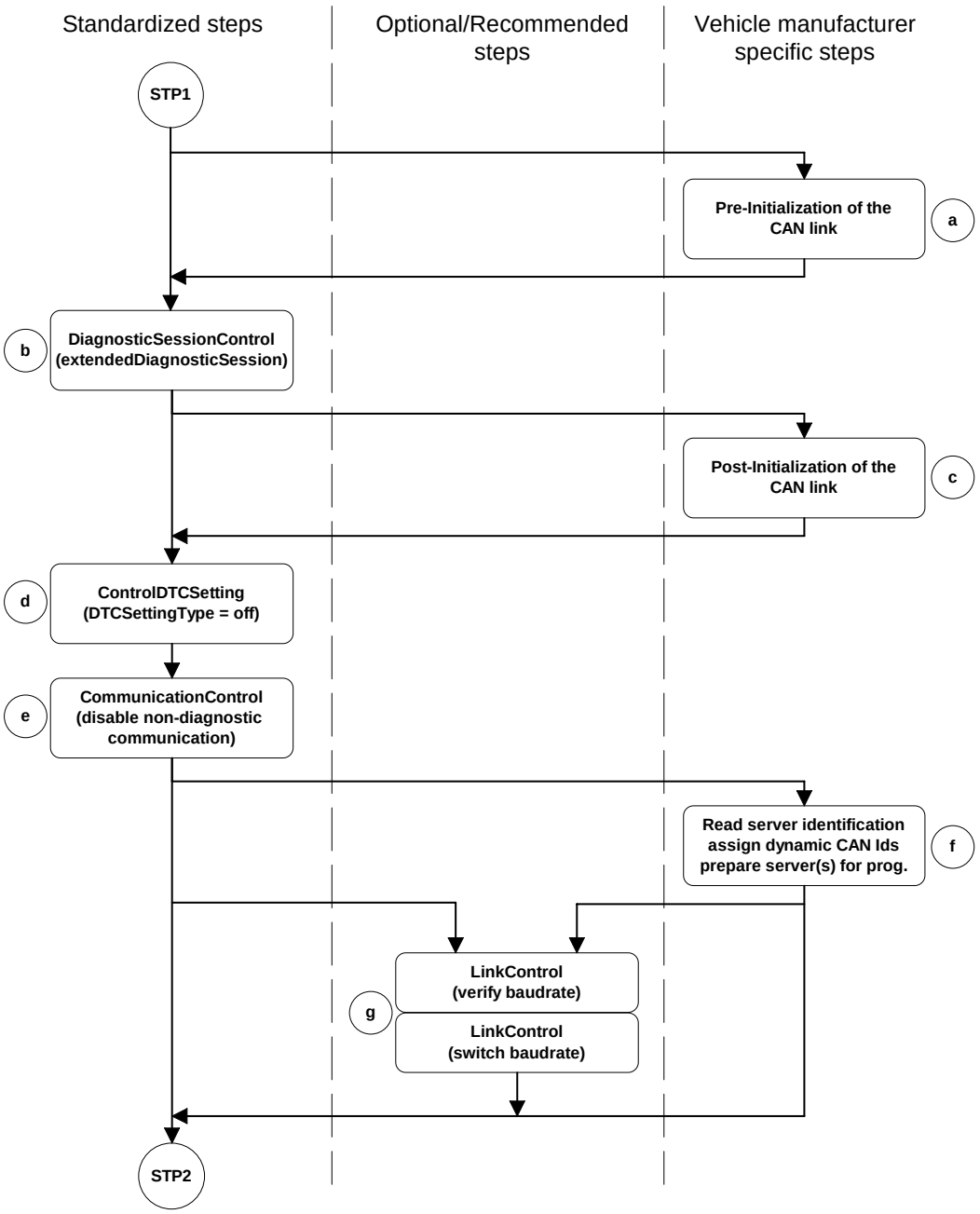

Figure 21 — Pre-Programming step of phase #1 (STP1)

a) **Pre-initialization of the CAN network:** Prior to any communication on the CAN link the network has to be initialized, which means that an initial wake-up of the CAN network has to be performed. The wake-up method and strategy is vehicle manufacturer specific and optionally to be used.

Furthermore this step allows for a determination of the server communication parameters such as the network configuration parameter serverDiagnosticAddress and the CAN identifiers used by the server(s).

b) **DiagnosticSessionControl (extendedDiagnosticSession**): In order to be able to disable the normal communication between the servers and the setting of DTCs it is required to start a non-defaultSession in each server where normal communication and DTCs shall be disabled. This is achieved via a DiagnosticSessionControl (10 hex) service with sessionType equal to extendedDiagnosticSession. The request is either transmitted

   — functionally addressed to all servers with a single request message, or

   — physically addressed to each server in a separate request message (requires a physically addressed TesterPresent (3E hex) request message to be transmitted to each server that is transitioned into a non-defaultSession).

   It is vehicle manufacturer specific whether response messages are required or not.

c) **Post-initialization of the CAN network:** Following the transition into the extendedDiagnosticSession further vehicle manufacturer specific CAN link initialisation steps can optionally be performed.

   Example: A vehicle manufacturer specific additional initialisation step can be to issue a request that causes gateway devices to perform a wake-up on all CAN links which are not accessible by the client directly through the diagnostic connector. The gateway will keep the CAN link(s) awake as long as the non-defaultSession is kept active in the gateway.

d) **ControlDTCSetting (DTCSettingType = off):** The client disables the setting of DTCs in each server using the ControlDTCSetting (85 hex) service with DTCSettingType equal to "off". The request is either transmitted

   — functionally addressed to all servers with a single request message, or

   — transmitted physically addressed to each server in a separate request message.

   It is vehicle manufacturer specific whether response messages are required or not.

e) **CommunicationControl (disable non-diagnostic communication):** The client disables the transmission and reception of non-diagnostic messages using the CommunicationControl (28 hex) service. The controlType parameter and communicationType parameter values are vehicle manufacturer specific (one OEM might disable the transmission only while another OEM might disable the transmission and the reception based on vehicle manufacturer specific needs). The request is either transmitted

   — functionally addressed to all servers with a single request message,

   — or transmitted physically addressed to each server in a separate request message.

   It is vehicle manufacturer specific whether response messages are required or not.

f) **Read server identification data / assign dynamic CAN Ids / prepare server(s) for programming:** After disabling normal communication an optional vehicle manufacturer specific step follows, which allows

   — reading the status of the server(s) to be programmed (e.g. application software/data programmed)

   — reading server identification data from the server(s) to be programmed

      — Identification (see ISO 14229-1 - recordDataIdentifier definitions): applicationSoftwareIdentification, applicationDataidentification, bootSoftwareIdentifiecation,

      — Fingerprint (see ISO 14229-1 - recordDataIdentifier definitions): applicationSoftwareFingerprint, applicationDataFingerprint, bootSoftwareFingerprint

— communication configuration such as dynamic assignment of CAN identifiers for a “Service ECU”,

— preparation of non-programmable servers for the upcoming programming event in order to allow them to optimise their CAN hardware acceptance filtering in a way that they can handle a 100% bus utilisation without dropping CAN frames (only accept the function request CAN identifier and its own physical request CAN identifier).

g) **LinkControl (verify baudrate/switch baudrate):** It is optional to increase the baudrate for the programming event in order to decrease the overall programming time and to gain additional bandwidth to be able to program multiple servers in parallel. A LinkControl (87 hex) service with linkControl equal to either verifyBaudrateTransitionWithFixedBaudrate or verifyBaudrateTransitionWithSpecificBaudrate is transmitted functionally or physically addressed to all servers with a single request message with responseRequired equal to "yes". This service is used to verify if a baudrate switch can be performed. At this point the baudrate switch is not performed. A second LinkControl (87 hex) service with sub-function transitionBaudrate is transmitted functionally addressed to all servers with a single request message with responseRequired equal to no.

Once the request message is successfully transmitted the client and all servers transition their baudrate to the previously verified baudrate for the programming event. The servers have to transition the baudrate within a vehicle manufacturer specific timing window. For this duration plus a safety margin the client is not allowed to transmit any request message onto the CAN network (including the TesterPresent request message). When the baudrate transition is successfully performed then the baudrate shall stay active for the duration the server switches between non-defaultSessions. Once the server transitions to the defaultSession it shall re-enable the normal speed baudrate of the CAN link it is connected to.

The usage of the baudrate switch requires the support of functional diagnostic communication in each server on a single CAN link that has to be transitioned to a higher baudrate, because the transition of the baudrate has to be performed at the same time by all nodes (incl. the client).

9.2.1.2 Programming step of phase #1 – download of application software and data

Following the pre-programming step the programming of one or multiple servers is performed. The programming sequence applies for a programming event of a single server and is therefore physically oriented. When multiple servers are programmed then multiple programming events either run in parallel or will be performed sequentially.

The following figure graphically depicts the functionality embedded in the programming step of phase #1. Following the figure a textual description can be found, which references the points in the figure.

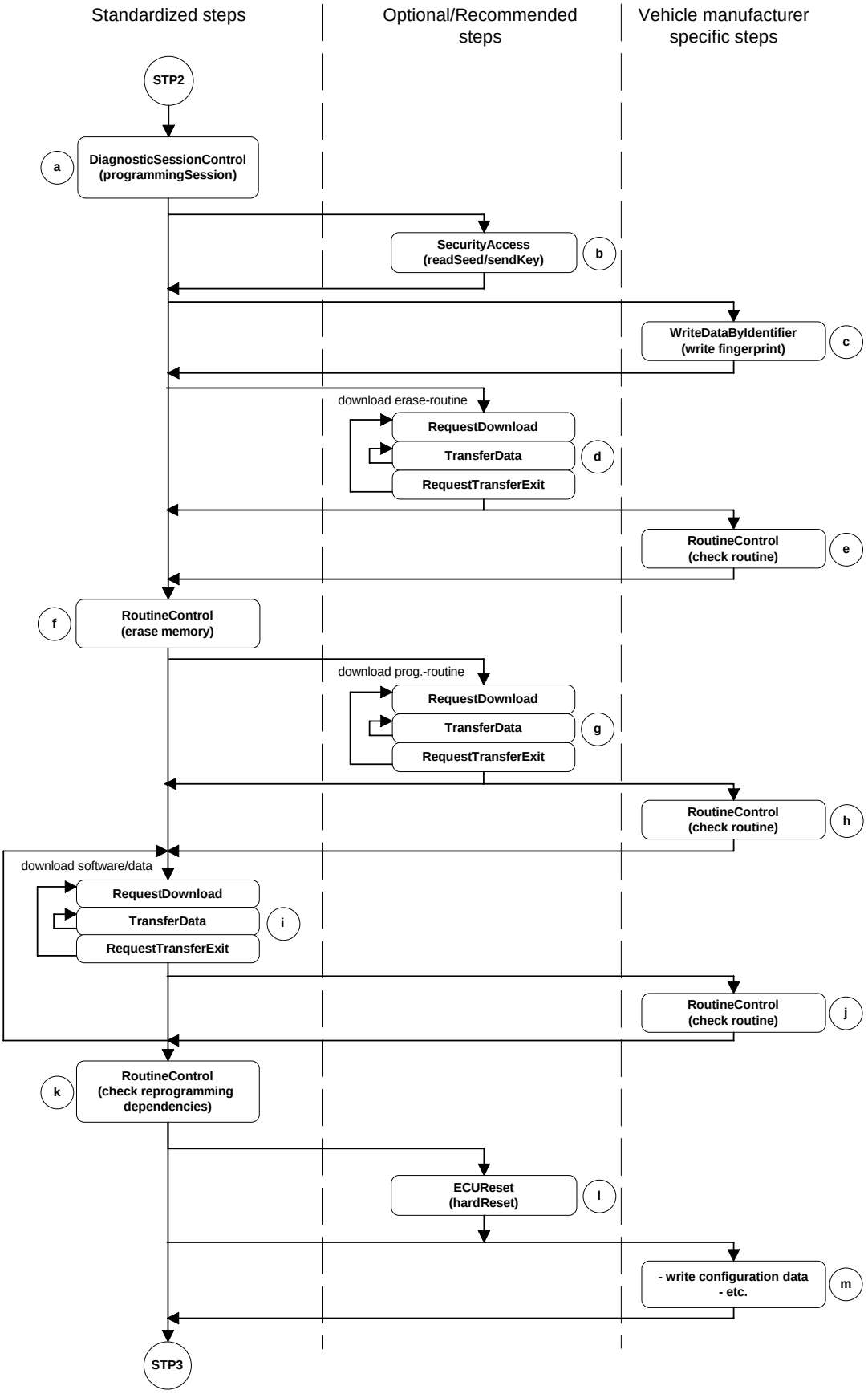

Figure 22 — Programming step of phase #1 (STP2)

a) **DiagnosticSessionControl (programmingSession):** The programming event is started in the server(s) via a physically/functionally addressed request of the DiagnosticSessionControl (10 hex) service with sessionType equal to programmingSession. When the server(s) receive(s) the request it/they shall allocate all necessary resources required for programming. It is implementation specific whether the server(s) start(s) executing out of boot memory.

b) **SecurityAccess (readSeed/sendKey):** It is recommended to secure a programming event. The SecurityAccess (27 hex) service shall be mandatory for emissions-related and safety systems. Other systems are not required to implement this service. The method on how a security access is performed is specified in ISO 14229-1.

c) **WriteDataByIdentifier (write fingerprint):** It is vehicle manufacturer specific to write a "fingerprint" into the server memory prior to the download of any routine and/or application software/data. The "fingerprint" identifies the one who modifies the server memory. When using this option then the recordDataIdentifiers bootSoftwareFingerprint, applicationSoftwareFingerprint and applicationDataFingerprint shall be used to write the fingerprint information (see ISO 14229-1 - recordDataIdentifier definitions).

d) **Download erase routine:** In case the server does not have the memory erase routine stored in permanent memory a download of the memory erase routine shall be performed. The download shall follow the specified sequence with RequestDownload (…), TransferData, and RequestTransferExit.

e) **RoutineControl (check routine):** It is vehicle manufacturer specific if a RoutineControl (31 hex) is used to check whether the download of the memory erase routine was successful. Alternative methods are to provide the result in the RequestTransferExit positive response message or via a negative response message including the appropriate negative response code to the RequestTransferExit request message.

f) **RoutineControl (erase memory):** The memory of the server shall be erased in order to allow an application software/data download. This is achieved via a routine, using the RoutineControl (31 hex) service to execute the erase routine.

g) **Download programming routine:** In case the server does not have the memory programming routine stored in permanent memory then a download of the memory programming routine has to be performed. The download shall follow the specified sequence with RequestDownload (34 hex), TransferData (36 hex), and RequestTransferExit (37 hex).

h) **RoutineControl (check routine):** It is vehicle manufacturer specific if a RoutineControl (31 hex) is used to check whether the download of the memory program routine was successful. Alternative methods are to provide the result in the RequestTransferExit positive response message or via a negative response message including the appropriate negative response code to the RequestTransferExit request message.

i) **Download application software/data:** Each download of a contiguous block of application software/data to a non-volatile server memory location (either a complete application software/data module or part of a software/data module) shall always follow the general data transfer method using the following service sequence:

   — RequestDownload (34 hex)

   — TransferData (36 hex)

   — RequestTransferExit (37 hex)

   A single application software/data block might require multiple TransferData (36 hex) request messages to be completely transmitted (this is the case if the length of the block exceeds the maximum network layer buffer size).

j) **RoutineControl (check routine):** It is vehicle manufacturer specific if a RoutineControl (31 hex) is used to check whether the download of the memory program routine was successful. Alternative methods are to provide the result in the RequestTransferExit positive response message or via a negative response message including the appropriate negative response code to the RequestTransferExit request message.

k) **RoutineControl (check programming dependencies):** Once all application software/data blocks/modules are completely downloaded the client shall verify if the download has been performed successfully by initiating a routine in the server using the RoutineControl (31 hex) service. This routine either triggers the server to check

— the reprogramming dependencies and to perform all necessary action to proof that the download and programming into non-volatile memory was successful, or

— it requests the server to calculate the checksum and submit the checksum to the client via a RoutineControl positive response message (requestRoutineResults) for a comparison with a checksum contained in the client.

The calculation method (e.g. CRC32, CRC16, 2 byte accumulated checksum, etc.) used is left to the discretion of the vehicle manufacturer.

The checksum comparison method (e.g. server side, client side) is left to the discretion of the vehicle manufacturer.

l) **ECUReset (hardReset):** Following the download of the application software/data it is optional to physically reset the re-programmed server in order to enable the downloaded application software/data. It has to be considered that the re-programmed server could activate a new set of diagnostic CAN identifiers, which differs to the ones used when performing the programming event (see 9.3). If either the server that was re-programmed does not change its communication parameters or the programming environment know the changed communication parameters then following the reset certain configuration data can be written to the re- programmed server.

m) **Write server configuration data:** Following the download of the application software/data it is vehicle manufacturer specific to perform further operations such as writing configuration data (e.g. VIN, etc.) back to the server. This also depends on the functionally that is supported by the re-programmed server when running out of boot.

9.2.1.3 Post-Programming step of phase #1 – re-synchronisation of CAN

The following figure graphically depicts the functionality embedded in the post-programming step of phase #1.

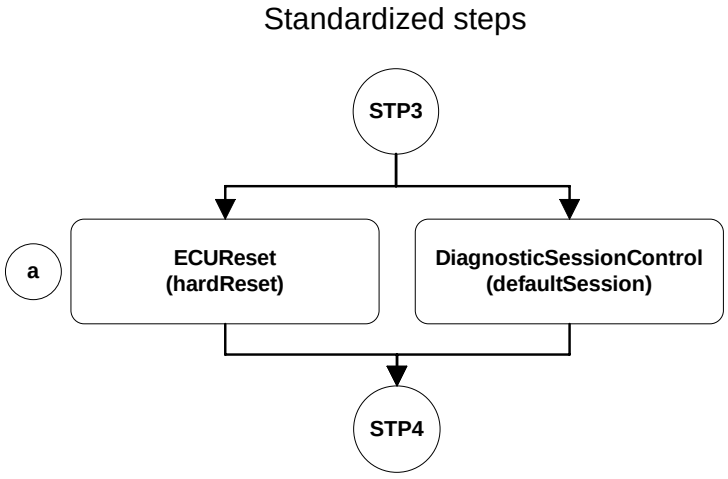

Figure 23 — Post-Programming step of phase #1 (STP3)

a) **ECUReset or DiagnosticSessionControl:** The client transmits either an ECUReset (11 hex) service request message onto the CAN network with resetType equal to hardReset or DiagnosticSessionControl (10 hex) with sessionType equal to defaultSession. This can either be done functionally addressed or physically addressed (depends on the supported vehicle approach). Further it is vehicle manufacturer specific whether a response message is required or not.

When a baudrate switch has been performed then this step has to be performed functionally, not requiring a response message, because the servers perform a baudrate transition to their normal speed of operation.

The reception of the ECUReset (11 hex) request message causes the server(s) to perform a reset and to start the defaultSession.

9.2.2 Programming phase #2 – server configuration

9.2.2.1 Pre-Programming step of phase #2 – server configuration

The pre-programming step of phase #2 is optional and should be used when there is the need to perform certain action after the software reset of the reprogrammed server. This will be the case when the server does not provide the required functionality to finally conclude the programming event when running out of boot during the programming step of phase #1.

The following figure graphically depicts the functionality embedded in the pre-programming step of phase #2. Following the figure a textual description can be found, which references the marks in the figure.

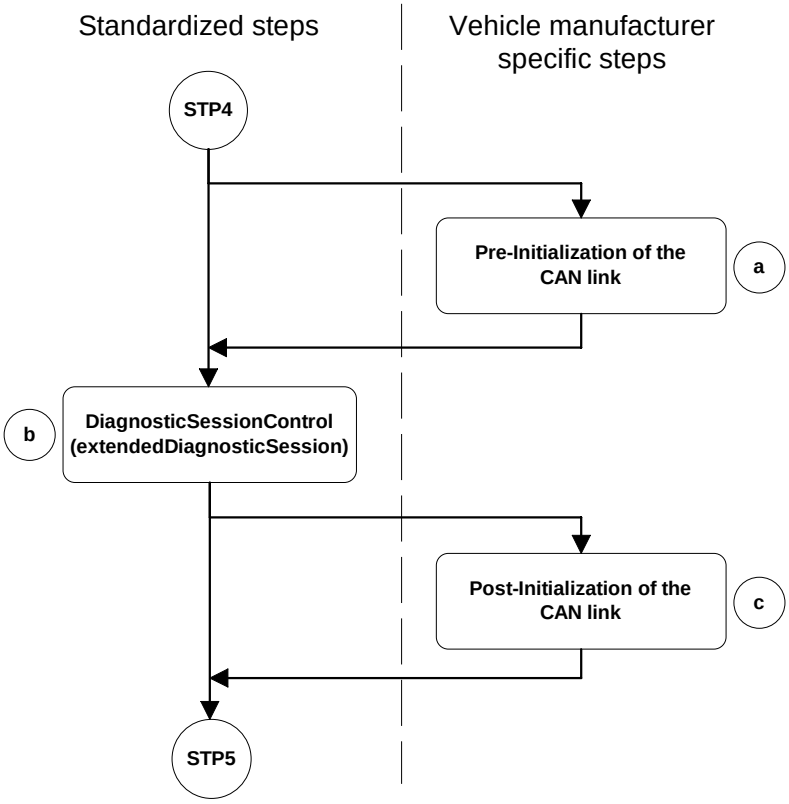

Figure 24 — Pre-Programming step of phase #2 (S4)

a) **Pre-initialisation of the CAN network:** Prior to any communication on the CAN link the network has to be initialised, which means that an initial wake-up of the CAN network has to be performed. The wake-up method and strategy is vehicle manufacturer specific and optionally to be used.

   Furthermore this step allows for a determination of the server communication parameters such as the network configuration parameter serverDiagnosticAddress and the CAN identifiers used by the server(s).

b) **DiagnosticSessionControl (extendedDiagnosticSession):** In order to be able to perform certain services in the programming step of phase #2 a non-defaultSession has to be started in each server on the CAN link that is involved in the conclusion of the programming event. This is performed via a DiagnosticSessionControl (10 hex) service with sessionType equal to extendedDiagnosticSession.

c) **Post-initialization of the CAN network:** Following the transition into the extendedDiagnosticSession further vehicle manufacturer specific CAN link initialisation steps can optionally be performed.

   Example: A vehicle manufacturer specific additional initialisation step can be to issue a request that causes gateway devices to perform a wake-up on all CAN links which are not accessible by the client directly through the diagnostic connector. The gateway will keep the CAN link(s) awake as long as the non-defaultSession is kept active in the gateway.

9.2.2.2 Programming step of phase #2 – final server configuration

The programming step of phase #2 is optional and contains any action that needs to take place with the reprogrammed server after the reset (when the application software is running) such as writing specific identification information. This step might be required in case the server does not provide the required functionality to perform an action when running out of boot during the programming step of phase #1. When multiple servers require performing additional functions then multiple programming steps can run in parallel or will be performed sequentially.

The following figure graphically depicts the functionality embedded in the post-programming step of phase #1.

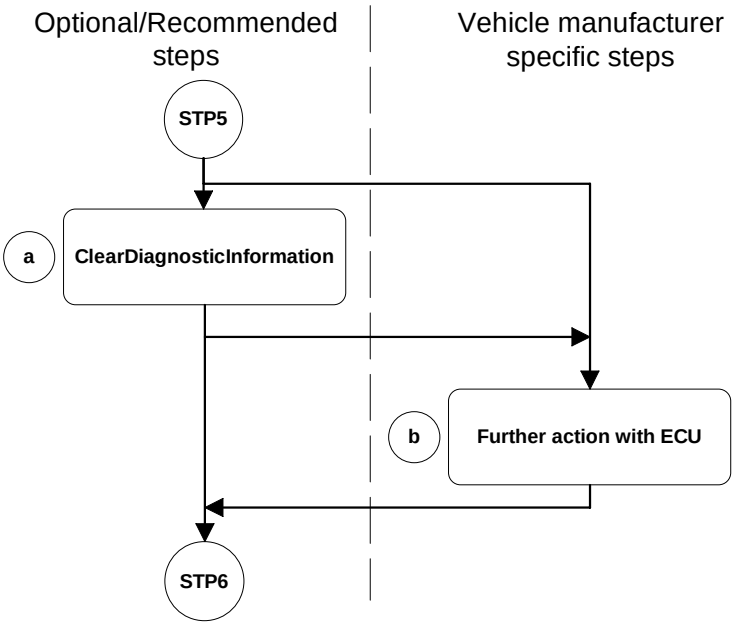

Figure 25 — Programming step #2 (STP5)

a) **ClearDiagnosticInformation:** In case the re-programmed server(s) has (have) been reset during the programming step of programming phase #1 then any diagnostic information that might have been stored in the re-programmed server(s) when the server was already running in the default session while other servers on the link still had normal communication disabled has to be reset via a physically addressed ClearDiagnosticInformation (14 hex) service.

b) **Further action with server:** The client performs any operation that is required in order to conclude the programming event with the server, such as writing configuration data (e.g. VIN).

9.2.2.3 Post-Programming step of phase #2 – re-synchronisation of CAN network

The post-programming step of phase #2 is used to conclude the programming phase #2.

The following figure graphically depicts the functionality embedded in the post-programming step of phase #2.

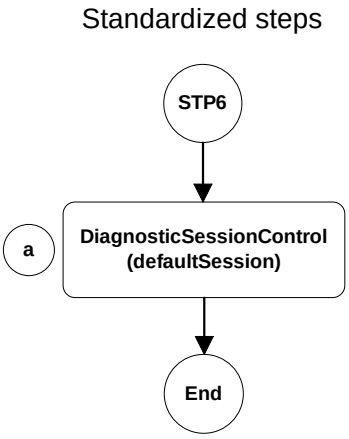

Figure 26 — Post-Programming step of phase #2 (STP6)

a) **DiagnosticSessionControl(defaultSession):** The client transmits a DiagnosticSessionControl (10 hex) request onto the CAN network with sessionType equal to defaultSession. The reception of the DiagnosticSessionControl (10 hex) causes all servers to start the defaultSession. The request can either be transmitted functionally addressed or physically addressed. The request shall be transmitted to all servers which were involved in the conclusion of the programming event and therefore stay in a non-defaultSession.

9.3 Requirements

The non-volatile server memory programming process can be applied to a programmable server in order to:

— Reprogram a server that has previously been fully programmed.

— Program a server which is shipped to the vehicle assembly plant or service facility without some element of it's full combination of application software and application data.

— Reprogram a server which has detected an error with memory locations containing software or calibration which forces the server to run out of boot software.

Programmable servers fall into three categories: PRG_TYPE_A, PRG_TYPE_B, or PRG_TYPE_C.

— **PRG_TYPE_A:**
A PRG_TYPE_A server is a programmable server that has already been fully programmed (either by the ECU supplier, the vehicle assembly plant, or in the service environment). PRG_TYPE_A servers are fully functional and can receive diagnostic requests and respond to them using the appropriate diagnostic CANId's. These diagnostic CANId's shall be further referenced as permanent diagnostic CANId's.

— **PRG_TYPE_B/PRG_TYPE_C:**
PRG_TYPE_B and PRG_TYPE_C servers are programmable servers that are missing some element of their full combination of application software and application data, or are executing boot software due to a memory error. A server which is missing application data (or is missing application software and application data) may, or may not, have permanent diagnostic CANId's pre-programmed. A programmable server which is not fully programmed and is used on a single platform would most likely have it's permanent diagnostic CANId's pre-programmed. A programmable server which is not fully programmed and can be used in multiple platforms, may not have the permanent diagnostic CANId's pre-programmed unless all of the platforms can standardise the CANId's used by that server (or multiple parts are released to accommodate the differences in CANId's between platforms). A PRG_TYPE_B server meets the above criteria and has its permanent diagnostic CANId's pre-programmed. A PRG_TYPE_C server meets the above criteria and does not have it's permanent diagnostic CANId's pre-programmed. PRG_TYPE_B and PRG_TYPE_C servers shall not attempt to participate in any non-diagnostic communication message exchange (inter-server communication).

NOTE A server executing boot software due to a memory fault is considered to be PRG_TYPE_B if permanent diagnostic CANId's are comprehended in the boot software. If the permanent diagnostic CANId's are not comprehended in boot software, then the server is considered PRG_TYPE_C.

If permanent diagnostic CANId's are pre-programmed, a programmable server shall respond to all diagnostic requests which contain one of the permanent diagnostic CANId's supported by the server (PRG_TYPE_A and PRG_TYPE_B). If permanent diagnostic CANId's are not pre-programmed (PRG_TYPE_C), the server shall not respond to diagnostic request messages until diagnostic responses are enabled. The support of PRG_TYPE_C servers and the process of enabling diagnostic responses are vehicle manufacturer specific.

The following is an example on how diagnostic responses can be enabled in PRG_TYPE_C servers.

Example for enabling diagnostic responses on PRG_TYPE_C servers:

A PRG_TYPE_C server shall not send positive or negative response messages for any diagnostic service until the client enables them. While diagnostic responses are disabled, the server shall only process (but not respond to) diagnostic requests sent using the functional request CANId addressed to all nodes on the CAN network. Diagnostic responses shall become enabled once the server receives a DiagnosticSessionControl (10 hex) service, followed by a CommunicationControl (28 hex) service request and a ReadDataByIdentifier (22 hex) service request with recordDataIdentifier equal to serverDiagnosticAddress. The PRG_TYPE_C server shall only respond to the ReadDataByIdentifier (22 hex) service request during this sequence, and shall respond to all subsequent diagnostic requests until a $S3_{Server}$ timeout occurs or an ECUReset (11 hex) request is received.

Once a PRG_TYPE_C server has enabled diagnostic responses, it shall enable two special case diagnostic CANId's for programming purposes. The special case CANId's are defined as:

— PRG_PrimeReq CANId = 0xx hex, where "xx" represents the server's diagnostic address value. This CANId shall be used as the physical request CANId.

— PRG_PrimeRsp CANId = 3xx hex, where "xx" represents the server's diagnostic address value. This CANId shall be used as the response CANId.

During the sequence to enable diagnostic responses, the PRG_TYPE_C server shall respond to the ReadDataByIdentifier (22 hex) service request using the PRG_PrimeRsp (3xx hex) CANId.

Diagnostic CANId's and non-diagnostic message CANId's for a fully programmed server are part of the application data downloaded to the server during a programming event. Upon completion of a software reset, the now completely programmed server(s) shall recognise its (their) specific CANId assignments for non-diagnostic and diagnostic messaging.

9.3.1 Requirements for all servers to support programming

During a programming session, servers shall default their physical I/O pins (wherever possible and without risk of damage to the server/vehicle and without risk of safety hazards) to a predefined state which minimises current draw.

Servers shall ensure that they can handle 100% bus utilisation at any allowed programming baud rate without dropping CAN frames during the programming event. A server may need to modify it's hardware acceptance filtering in order to meet this requirement (see also section 9.2.1.1). The server(s) actually being programmed can recognise the need to modify its acceptance filtering by receiving the DiagnosticSessionControl (10 hex) service with sessionType equal to programmingSession. The server(s) which are not actually programmed (programmable or non-programmable servers) might need to be instructed to modify their acceptance filtering for the duration of the programming event. The method on how those servers are instructed is vehicle manufacturer specific.

The DiagnosticSessionControl (10 hex) service with sessionType equal to extendedDiagnosticSession followed by a CommunicationControl (28 hex) service that disables non-diagnostic messages and a ControlDTCSetting (85 hex) service that disables setting of DTCs shall be used by the server to recognise the disabling of normal communication and to ensure that a server does not set DTC's while another server is being programmed. The extendedDiagnosticSession has to be kept active by the client for the duration of the programming event.

9.3.2 Requirements for programmable servers to support programming

9.3.2.1 Hardware Requirements

All servers that are programmable must be able to interface with the programming tools used by development, the assembly plant, and by service via the appropriate pins of the vehicle diagnostic connector. The only power required at the vehicle diagnostic connector for programming shall be vehicle battery power.

Any server that is properly installed in the vehicle and is programmable shall be able to be programmed via the vehicle diagnostic connector. It shall not be required to remove the server from the vehicle in order to perform programming.

9.3.2.2 Software requirements

9.3.2.2.1 Application software

If the application software is programmable, then the application software shall be capable of being programmed separately from the application data. This allows for assembly plant programming of only calibrations. Deviations from this requirement must be agreed upon by all responsible at the OEM and shall be documented accordingly.

A server shall be capable of using the same diagnostic CANId's for the duration of a programming event. This means that a server which stores the permanent diagnostic CANId's in calibration and is fully programmed at the beginning of a programming event (SPS_TYPE_A server), shall use the permanent diagnostic CANId's even after the application data have been erased. The application software is not required to retain the permanent diagnostic CANId's if the programming event is interrupted prior to its completion and the server has performed a software reset.

A server that is only capable of application data programming (e.g., the application software is part of ROM and application date stored in EEPROM) shall have in the application software the equivalent functionality specified for the boot software in the subsequent sections of this chapter.

9.3.2.2.2 Application data

Application data (calibration data) shall be capable of being programmed separately from the application software. This allows for assembly plant programming of only application date. Deviations from this requirement must be agreed upon by the responsible at the OEM and shall be documented accordingly.

The server shall support either one or both of the following methods of programming calibrations.

— Programming an individual application data module.

— Programming multiple (or all) of the application data modules during a single programming event.

9.3.2.3 Boot software description and requirements

All programmable servers that support programming of the application software shall contain boot software in a boot memory region. Servers that support boot software shall continue to execute out of the boot until a complete set of application software and application date is programmed.

The boot memory shall be protected against inadvertent erasure such that a failed attempt to modify application data or application software does not prohibit the server's ability to recover and be programmed after the failed attempt. The server shall be able to recover and be reprogrammed if any of the following error conditions occur during the programming process.

— Loss of supplied power connection.

— Loss of the ground connection.

— Disruption of CAN communication.

— Over or under voltage conditions.

Boot software resides in the boot memory region and is the software that a server begins executing upon power-up. Transfer of program control to the boot software also occurs once the server is informed that it is about to be programmed (reference the DiagnosticSessionControl service and the programming process defined in section 9.2.1.2). All Programmable servers operating out of Boot memory shall not transmit any non-diagnostic communication messages or any unsolicited diagnostic messages.

9.3.2.3.1 Boot Software General Requirements

All servers operating out of boot memory shall be able to receive diagnostic messages. A server shall be capable of using the same diagnostic CANId’s for the duration of a programming event. This means that a server which is fully programmed at the beginning of a programming event (PRG_TYPE_A server), shall use it’s permanent diagnostic CANId’s during the programming. To accomplish this, the permanent diagnostic CANId’s must be provided to the boot software from the application software when program control is transferred back to the boot software. The boot software is not required to retain the permanent diagnostic CANId’s passed from the application software if the programming event is interrupted prior to it’s completion and the server has performed a software reset and becomes a PRG_TPE_C server (this is valid if the boot software only supports the non-permanent CANId's).

The boot software shall be protected. The boot software can be protected via hardware (e.g. via settings in a control register which prevents certain sectors of the memory from being erased or written to) or software (e.g. address range restrictions in the programming routines). It is recommended that the boot software is not capable of being modified by the same programming erase/write routines that are used to modify the application software and application date. Programming the boot software as part of the programming process

may be allowed provided that a mechanism is in place to ensure that there is no possibility that the server could fail at a point in of the programming process where it cannot recover and be programmed with a subsequent programming event.

9.3.2.3.2 Boot software diagnostic service requirements

During the post-programming step of phase #1 the server either runs out of application software or out of boot, while during the programming step of phase #1 the server runs completely out of boot, because it has transitioned to boot when the programming session is enabled via the DiagnosticSessionControl (10 hex) service.

During programming phase #2 the application software is running and therefore the full diagnostic functionality of the server is available, which is a superset of the boot software diagnostic service requirements.

The following tables define the minimum diagnostic service requirements for the boot software of a programmable server. The listed services have to supported in order to fulfil the requirements for performing non-volatile server memory programming during programming phase #1.

The tables make use of the steps defined for programming phase #1 (see section 9.2.1).

Table 51 — Boot software diagnostic service support during pre-programming step of phase #1

Service	sub-function / data parameter	Sequence step #	Remark
DiagnosticSessionControl (10 hex)	sessionType = extendedDiagnosticSession (03 hex)	(b)	**Mandatory:** Required for session management ($S3_{Server}$ timeout, especially when performing a baudrate transition and SecurityAccess service).
CommunicationControl (28 hex)	controlType = vehicle manufacturer specific (disable non-diagnostic communication messages)	(d)	**Mandatory:** The server does not need to perform any special action (non-diagnostic messages are disabled when running out of boot), except the transmission of a positive response message.
ControlDTCSetting (85 hex)	DTCSettingType = off (02 hex)	(e)	**Mandatory:** The server does not need to perform any special action (DTCs are disabled when running out of boot), except the transmission of a positive response message.
ReadDataByIdentifier (22 hex)	recordDataIdentifier = vehicle manufacturer specific	(f)	**Optional:** Required to be supported when reading software/data identification data.
LinkControl (87 hex)	linkControlType = verifyWithFixedBaudrate (01 hex), verifyWithSpecifcBaudrate (02 hex), transitionBaudrate (03 hex)	(g)	**Optional:** Required to be supported when performing a baudrate switch.

The service(s) to be supported for step (a), (c) and (f) have to be defined by the vehicle manufacturer.

Table 52 — Boot software diagnostic service support during programming step of phase #1

Service	sub-function / data parameter	Sequence step #	Remark
DiagnosticSessionControl (10 hex)	sessionType = programmingSession (02 hex)	(a)	**Mandatory:** Required for compatibility with application software in order to allow for the identical handling in the programming application of the client.
SecurityAccess (27 hex)	securityAccessType = readSeed (01 hex), sendKey (03 hex)	(b)	**Optional:** Required to be supported by theft, emission and safety related systems.
WriteDataByIdentifier (2E hex)	bootSoftwareFingerprint, appSoftwareFingerprint, appDataFingerprint, vehicle manufacturer specific	(c), (m)	**Optional:** Required for writing the fingerprint and other identification data.
RequestDownload (34 hex)	vehicle manufacturer specific	(d), (g), (i)	**Mandatory:** In general required for the transfer of data from the client to the server when running out of boot.
TransferData (36 hex)	routine data, application software, or application data		
RequestTransferExit (37 hex)	vehicle manufacturer specific		
RoutineControl (31 hex)	routineControlType = startRoutine (01 hex) routineIdentifier = checkProgDependencies	(e), (f), (h), (j), (k)	**Mandatory:** Required for the check of the programming dependencies. Can also be used when an optional check of a successful transfer of data is performed (on a service 34, 36, 37 hex sequence basis).
ECUReset (11 hex)	resetType = hardReset (01 hex)	(l)	**Mandatory:** Required for a physical reset of the re-programmed server at the end of the programming step. The server(s) that have been reprogrammed are forced to perform a software reset in order to start the application software.

The service(s) to be supported for step (m) have to be defined by the vehicle manufacturer.

Table 53 — Boot software diagnostic service support during post-programming step of phase #1

Service	sub-function / data parameter	Sequence step #	Remark
ECUReset (11 hex)	hardReset	(a)	**Mandatory:** The server(s) that have been reprogrammed are forced to perform a software reset in order to start the application software.

9.3.2.4 Security requirements

All programmable servers that have emission, safety, or theft related features shall employ a seed and key security feature, accessible via the SecurityAccess (27 hex) service, to protect the programmed server from inadvertent erasure and unauthorised programming. All field service replacement servers that meet the above criteria, shall be shipped to the field with the security feature activated (i.e. a programming tool cannot gain access to the server without first gaining access through the SecurityAccess service).

All development servers shall use the 1's complement of the seed as the valid key.

9.3.2.5 Application software and application data file requirements

9.3.2.5.1 File formats

The boot software, application software and application data files can have the following formats:

— **binary – raw binary file**

This means that the client issues a RequestDownload (34 hex) service prior to the download, transmits the whole data of the binary file with one or multiple TransferData (36 hex) services, and concludes the transfer via the RequestTransferExit (37 hex) service.

— **Intel Hex – ASCII file according to Intel Hex format**

This means that the client transfers each contiguous block of data contained in the Intel Hex file by issuing a RequestDownload (34 hex) service prior to the download of the contiguous block, transmits the whole data of the contiguous block with one or multiple TransferData (36 hex) services, and concludes the transfer via the RequestTransferExit (37 hex) service. The client repeats the transfer sequence until all contiguous blocks of the Intel Hex file are transferred.

— **Motorola S19– ASCII file according to Motorola S19 format**

This means that the client transfers each contiguous block of data contained in the Motorola S19 file by issuing a RequestDownload (34 hex) service prior to the download of the contiguous block, transmits the whole data of the contiguous block with one or multiple TransferData (36 hex) services, and concludes the transfer via the RequestTransferExit (37 hex) service. The client repeats the transfer sequence until all contiguous blocks of the Motorola S19 file are transferred.

It shall be avoided to transfer blocks of data during the programming step where the non-volatile memory is not affected by the transmitted data. This either requires the use of the Intel Hex or Motorola S19 file format or a split into multiple modules when binary file format is used.

9.3.3 Software and data identification and fingerprints

9.3.3.1 Software and data identification

The boot software, application software and application data shall be identified via the following recordDataIdentifiers (see also ISO 14229-1):

Table 54 — Software and data identification

Hex	Description	Mnemonic
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.	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.	ASI

Table 54 — Software and data identification

Hex	Description	Mnemonic
F182	**applicationDataIdentificatioin** 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.	ADI

The above given recordDataidentifier definitions result in the following structure of the data portion of each recordDataIdentifier.

Table 55 — bootSoftwareIdentification, applicationSoftwareIdentification and applicationDataIdentification data record definitions

Byte pos. in record	Description	Cvt	Hex Value	Mnemonic
#1	numberOfModules	M	00-FF	NOM
	identificationParameterRecord[] #1 = [			IDPREC_
#2	data#1	C_1	00-FF	DATA_1
:	:	:	:	:
#m+1	data#m]	C_1	00-FF	DATA_m
	:			
	identificationParameterRecord[] #k = [			IDPREC_
#n-m	data#1	C_2	00-FF	DATA_1
:	:	:	:	:
#n	data#m]	C_2	00-FF	DATA_m

C_1: This parameter is present when at least one identificationParameterRecord is available to be reported (numberOfModules greater or equal than one (1)).

C_2: This parameter is present when more than one identificationParameterRecord is available to be reported (numberOfModules greater than one (1)).

The structure of the identificationParameterRecord for bootSoftwareIdentification, applicationSoftwareIdentification, and applicationDataIdentificatioin is vehicle manufacturer specific and shall be the same for all identification information.

In case no application data or application software is programmed in the server then numberOfModules equal to zero (0) shall be reported and no identificationParameterRecord shall be present.

NOTE In case a server supports multiple identicationParameterRecords for either application software, application data or boot software then its network layer must be capable to transmit the multi-frame response message. This does not necessarily require that the network layer have to reserve the maximum buffer for the transmission of this multi-frame message. The transmission of a long multi-frame message can also be handled via a ring buffer implementation. The application dynamically feeds the network layer with data while the transmission of the multi-frame message is in progress. This kind of implementation allows the transmission of long multi-frame messages by keeping the required RAM buffer small.

The bootSoftwareIdentification, applicationSoftwareIdentification and applicationDataIdentification shall be part of each module that is downloaded into the server, therefore any write operation to the defined recordDataIdentifiers shall be rejected by the server.

The following is an example on how the identificationParameterRecord could be structured. The example assumes that the server reports two (2) identificationParameterRecords.

NOTE This example shall not to be treated as a recommendation for an implementation!

Table 56 — Example for an identificationParameterRecord structure

Byte pos. in record	Description		Hex Value	Mnemonic
#1	numberOfModules		02	NOM
	identificationParameterRecord[] #1 = [			IDPREC_
#2		moduleId	01	DATA_1
#3		version (high byte)	02	DATA_2
#4		version (low byte)	17	DATA_3
#5		partNumber (byte #1)	64	DATA_4
#6		partNumber (byte #2)	37	DATA_5
#7		partNumber (byte #3)	45	DATA_6
#8		partNumber (byte #4)]	82	DATA_7
	identificationParameterRecord[] #2 = [			IDPREC_
#9		moduleId	02	DATA_1
#10		version (high byte)	35	DATA_2
#11		version (low byte)	00	DATA_3
#12		partNumber (byte #1)	64	DATA_4
#13		partNumber (byte #2)	37	DATA_5
#14		partNumber (byte #3)	63	DATA_6
#15		partNumber (byte #4)]	84	DATA_7

9.3.3.2 Software and data fingerprints

A fingerprint uniquely identifies the programming tool that erased and/or reprogrammed the server software/data. If the server software/data is separated in several modules, the fingerprint could also identify which software/data module is manipulated (e.g. boot software, application software, and application data). A fingerprint has to be written into non-volatile memory of the server before any software/data manipulation occurs (e.g. before erasing the flash memory).

The boot software, application software and application data fingerprints shall be identified via the following recordDataIdentifiers (see also ISO 14229-1):

Table 57 — Software and data fingerprint identification

Hex	Description	Mnemonic
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.	BSFP
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.	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.	ADFP

The above given recordDataIdentifier definitions result in the following structure of the data portion of each recordDataIdentifier.

Table 58 — bootSoftwareFingerprint, applicationSoftwareFingerprint and applicationFingerprint data record definitions

Byte pos. in record	Description	Cvt	Hex Value	Mnemonic
	fingerprintParameterRecord[] = [			FPPREC_
#1	data#1	M	00-FF	DATA_1
:	:	:	:	:
#n	data#m]	U	00-FF	DATA_m

The structure of the fingerprintParameterRecord for bootSoftwareFingerprint, applicationSoftwareFingerprint, and applicationDataFingerprint is vehicle manufacturer specific and shall be the same for all fingerprint information.

The following is an example on how the fingerprintParameterRecord could be structured. The given fingerprint belongs to the module with module Id 02 hex.

NOTE This example shall not to be treated as a recommendation for an implementation!

Table 59 — Example for a fingerprintParameterRecord structure

Byte pos. in record	Description	Hex Value	Mnemonic
	fingerprintParameterRecord[] #1 = [		IDPREC_
#2	moduleId	02	DATA_1
#3	TesterSerialNumber (byte #1)	02	DATA_2
#4	TesterSerialNumber (byte #2)	17	DATA_3
#5	TesterSerialNumber (byte #3)	22	DATA_4
#6	TesterSerialNumber (byte #4)	64	DATE_5
#7	programmingDate (byte #1)	37	DATA_4
#8	programmingDate (byte #2)	45	DATA_5
#9	programmingDate (byte #3)]	82	DATA_6

9.3.4 Server routine access

Routines are used to perform non-volatile memory access such as erasing non-volatile memory and checking the successful download of a module.

The following table contains the routineIdentifiers defines for non-volatile memory access (see also ISO 14229-1):

Table 60 — routineIdentifiers for non-volatile memory access

Hex	Description	Mnemonic
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.	EM
FF01	**checkProgrammingDependendcies** 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.	CPD

The checkProgrammingDependencies routineIdentifier can for example be used to either transfer an already calculated checksum to the server for a server-internal check or to request the calculation of the checksum in the server and to provide the calculated values in the response message for a client internal check. Furthermore the checkProgrammingDependencies routineIdentifier can for example be used to trigger a programming check routine in the server, which only indicates success or not and optionally could provide results such as the error reason by using the appropriate negative response code or appropriate results values in the positive response message.

There are the following alternatives for the activation of the routine in the server:

a) The client transmits a RoutineControl (31 hex) service request message to the server. The server accepts the request and indicates this via a positive response message. The requested routine is started in the background, therefore the results are not yet provided in the positive response message. The client has to request the routine results via the RoutineControl (31 hex) service (polling of routine results).

b) The client transmits a RoutineControl (31 hex) service request message to the server. The server accepts the request and starts the routine. The server does not send a positive response until the routine is stopped. In order to keep the communication with the client active the server may need to transmit negative response messages including response code 78 hex (requestCorrectlyReceived-ResponsePending). When the routine is completely executed then the server transmits the RoutineControl positive response message that includes the results of the performed check of the programming dependencies.

9.4 Non-volatile server memory programming message flow examples

This section shows the message for a non-volatile server memory-programming event of a single server. The given message flows are based on a single server and the transfer of four (4) modules, where each module has a length of 511 bytes (data bytes 02 hex through FE hex). The network layer buffer size of the server that is re-programmed is 255 bytes (reported in the RequestDownload positive response message).

9.4.1 Programming phase #1 - Pre-Programming step

Table 61 — StartDiagnosticSession(extendedSession)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
27.2174	1	7DF	Request	3	02 10 03	STDS message
0.0002	1	7E9	Response	7	06 50 03 00 96 17 70	STDS message

Table 62 — ControlDTCSetting(off)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.0505	1	7DF	Request	3	02 85 02	CDTCS message
0.0001	1	7E8	Response	3	02 C5 02	CDTCS message
0.0001	1	7E9	Response	3	02 C5 02	CDTCS message

Table 63 — CommunicationControl(disble normal communication)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.0007	1	7DF	Request	4	03 28 03 03	CC message
0.0001	1	7E8	Response	3	02 68 03	CC message
0.0001	1	7E9	Response	3	02 68 03	CC message

9.4.2 Programming phase #1 - Programming step

Table 64 — StartDiagnosticSession(programmingSession)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.6964	1	7DF	Request	3	02 10 02	STDS message
0.0012	1	7E8	Response	7	06 50 02 00 FA 0B B8	STDS message

Table 65 — Interleaved TesterPresent

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.9987	1	7DF	Request	3	02 3E 80	TP message

Table 66 — SecurityAccess(readSeed)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.0000	1	7E0	Request	3	02 27 01	SA message
0.9989	1	7DF	Request	3	02 3E 80	TP message

Table 67 — SecurityAccess(sendKey)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.9998	1	7E0	Request	5	04 27 02 47 11	SA message
0.0002	1	7DF	Request	3	02 3E 80	TP message
0.0008	1	7E8	Response	3	02 67 02	SA message
1.9992	1	7DF	Request	3	02 3E 80	TP message

Table 68 — RoutineControl(eraseMemory)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.9995	1	7E0	Request	5	04 31 01 FF 00	RC message
0.0001	1	7E8	Response	4	03 7F 31 78	NR w/ NRC78
1.0004	1	7DF	Request	3	02 3E 80	TP message
1.9995	1	7E8	Response	4	03 7F 31 78	NR w/ NRC78
0.0005	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message
1.0002	1	7E8	Response	5	04 71 01 FF 00	RC message
0.9998	1	7DF	Request	3	02 3E 80	TP message

Table 69 — RequestDownload - Module #1

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.9989	1	7E0	Request	8	34 00 33 00 19 68	RD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0010	1	7DF	Request	3	02 3E 80	TP message
0.0001	1	7E0	Request	4	21 00 01 FF	RD message (CF)
0.0012	1	7E8	Response	5	04 74 20 00 FF	RD message
1.9987	1	7DF	Request	3	02 3E 80	TP message

Table 70 — TransferData – Module #1 (block #1)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.9996	1	7E0	Request	8	10 FF 36 01 02 03 04 05	TD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0012	1	7E0	Request	8	21 06 07 08 09 0A 0B 0C	TD message (CF)
0.0010	1	7E0	Request	8	22 0D 0E 0F 10 11 12 13	TD message (CF)
0.0010	1	7E0	Request	8	23 14 15 16 17 18 19 1A	TD message (CF)
:	:	:	:	:	:	:
0.0010	1	7E0	Request	8	23 F4 F5 F6 F7 F8 F9 FA	TD message (CF)
0.0009	1	7E0	Request	5	24 FB FC FD FE	TD message (CF)
0.0011	1	7E8	Response	3	02 76 01	TD message
0.9630	1	7DF	Request	3	02 3E 80	TP message

Table 71 — TransferData – Module #1 (block #2)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.9994	1	7E0	Request	8	10 FF 36 02 02 03 04 05	TD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0012	1	7E0	Request	8	21 06 07 08 09 0A 0B 0C	TD message (CF)
0.0010	1	7E0	Request	8	22 0D 0E 0F 10 11 12 13	TD message (CF)
0.0010	1	7E0	Request	8	23 14 15 16 17 18 19 1A	TD message (CF)
:	:	:	:	:	:	:
0.0010	1	7E0	Request	8	23 F4 F5 F6 F7 F8 F9 FA	TD message (CF)
0.0009	1	7E0	Request	5	24 FB FC FD FE	TD message (CF)
0.0011	1	7E8	Response	3	02 76 02	TD message
1.9633	1	7DF	Request	3	02 3E 80	TP message

Table 72 — TransferData – Module #1 (block #3)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.9991	1	7E0	Request	8	07 36 03 02 03 04 05 06	TD message (FF)
0.0011	1	7E8	Response	3	02 76 03	TD message
0.9998	1	7DF	Request	3	02 3E 80	TP message

Table 73 — RequestTransferExit - Module #1

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.9999	1	7E0	Request	2	01 37	RTE message
0.0002	1	7DF	Request	3	02 3E 80	TP message
0.0009	1	7E8	Response	2	01 77	RTE message
1.9992	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message

Table 74 — RequestDownload - Module #2

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.9995	1	7E0	Request	8	10 09 34 00 33 00 1B 67	RD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0004	1	7DF	Request	3	02 3E 80	TP message
0.0007	1	7E0	Request	4	21 00 01 FF	RD message (FF)
0.0012	1	7E8	Response	5	04 74 20 00 FF	RD message
1.9982	1	7DF	Request	3	02 3E 80	TP message

Table 75 — TransferData – Module #2 (block #1)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.0002	1	7E0	Request	8	10 FF 36 01 02 03 04 05	TD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0012	1	7E0	Request	8	21 06 07 08 09 0A 0B 0C	TD message (CF)
0.0010	1	7E0	Request	8	22 0D 0E 0F 10 11 12 13	TD message (CF)
0.0010	1	7E0	Request	8	23 14 15 16 17 18 19 1A	TD message (CF)
:	:	:	:	:	:	:
0.0010	1	7E0	Request	8	23 F4 F5 F6 F7 F8 F9 FA	TD message (CF)
0.0009	1	7E0	Request	5	24 FB FC FD FE	TD message (CF)
0.0011	1	7E8	Response	3	02 76 01	TD message
1.9626	1	7DF	Request	3	02 3E 80	TP message

Table 76 — TransferData – Module #2 (block #2)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.9994	1	7E0	Request	8	10 FF 36 02 02 03 04 05	TD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0012	1	7E0	Request	8	21 06 07 08 09 0A 0B 0C	TD message (CF)
0.0010	1	7E0	Request	8	22 0D 0E 0F 10 11 12 13	TD message (CF)
0.0010	1	7E0	Request	8	23 14 15 16 17 18 19 1A	TD message (CF)
:	:	:	:	:	:	:
0.0010	1	7E0	Request	8	23 F4 F5 F6 F7 F8 F9 FA	TD message (CF)
0.0009	1	7E0	Request	5	24 FB FC FD FE	TD message (CF)
0.0011	1	7E8	Response	3	02 76 02	TD message
1.9633	1	7DF	Request	3	02 3E 80	TP message

Table 77 — TransferData – Module #2 (block #3)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.9996	1	7E0	Request	8	07 36 03 02 03 04 05 06	TD message (FF)
0.0011	1	7E8	Response	3	02 76 03	TD message
0.9993	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message

Table 78 — RequestTransferExit - Module #2

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.0002	1	7E0	Request	2	01 37	RTE message
0.0011	1	7E8	Response	2	01 77	RTE message
1.9987	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message

Table 79 — RequestDownload - Module #3

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.9995	1	7E0	Request	8	10 09 34 00 33 00 1D 66	RD message (FF)
0.0001	1	7DF	Request	3	02 3E 80	TP message
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0011	1	7E0	Request	4	21 00 01 FF	RD message (FF)
0.0012	1	7E8	Response	5	04 74 20 00 FF	RD message
1.9976	1	7DF	Request	3	02 3E 80	TP message

Table 80 — TransferData – Module #3 (block #1)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.0006	1	7E0	Request	8	10 FF 36 01 02 03 04 05	TD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0012	1	7E0	Request	8	21 06 07 08 09 0A 0B 0C	TD message (CF)
0.0010	1	7E0	Request	8	22 0D 0E 0F 10 11 12 13	TD message (CF)
0.0010	1	7E0	Request	8	23 14 15 16 17 18 19 1A	TD message (CF)
:	:	:	:	:	:	:
0.0010	1	7E0	Request	8	23 F4 F5 F6 F7 F8 F9 FA	TD message (CF)
0.0009	1	7E0	Request	5	24 FB FC FD FE	TD message (CF)
0.0011	1	7E8	Response	3	02 76 01	TD message
1.9619	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message

Table 81 — TransferData – Module #3 (block #2)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.0003	1	7E0	Request	8	10 FF 36 02 02 03 04 05	TD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0012	1	7E0	Request	8	21 06 07 08 09 0A 0B 0C	TD message (CF)
0.0010	1	7E0	Request	8	22 0D 0E 0F 10 11 12 13	TD message (CF)
0.0010	1	7E0	Request	8	23 14 15 16 17 18 19 1A	TD message (CF)
:	:	:	:	:	:	:
0.0010	1	7E0	Request	8	23 F4 F5 F6 F7 F8 F9 FA	TD message (CF)
0.0009	1	7E0	Request	5	24 FB FC FD FE	TD message (CF)
0.0011	1	7E8	Response	3	02 76 02	TD message
1.9622	1	7DF	Request	3	02 3E 80	TP message

Table 82 — TransferData – Module #3 (block #3)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.0002	1	7E0	Request	8	07 36 03 02 03 04 05 06	TD message (FF)
0.0011	1	7E8	Response	3	02 76 03	TD message
0.9998	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message

Table 83 — RequestTransferExit - Module #3

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.0007	1	7E0	Request	2	01 37	RTE message
0.0011	1	7E8	Response	2	01 77	RTE message
1.9982	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message

Table 84 — RequestDownload - Module #4

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.0004	1	7E0	Request	8	10 09 34 00 33 00 1F 65	RD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0011	1	7E0	Request	4	21 00 01 FF	RD message (FF)
0.0012	1	7E8	Response	5	04 74 20 00 FF	RD message
1.9972	1	7DF	Request	3	02 3E 80	TP message

Table 85 — TransferData – Module #4 (block #1)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.0012	1	7E0	Request	8	10 FF 36 01 02 03 04 05	TD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0012	1	7E0	Request	8	21 06 07 08 09 0A 0B 0C	TD message (CF)
0.0010	1	7E0	Request	8	22 0D 0E 0F 10 11 12 13	TD message (CF)
0.0010	1	7E0	Request	8	23 14 15 16 17 18 19 1A	TD message (CF)
:	:	:	:	:	:	:
0.0010	1	7E0	Request	8	23 F4 F5 F6 F7 F8 F9 FA	TD message (CF)
0.0009	1	7E0	Request	5	24 FB FC FD FE	TD message (CF)
0.0011	1	7E8	Response	3	02 76 01	TD message
1.9614	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message

Table 86 — TransferData – Module #4 (block #2)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.0009	1	7E0	Request	8	10 FF 36 02 02 03 04 05	TD message (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0012	1	7E0	Request	8	21 06 07 08 09 0A 0B 0C	TD message (CF)
0.0010	1	7E0	Request	8	22 0D 0E 0F 10 11 12 13	TD message (CF)
0.0010	1	7E0	Request	8	23 14 15 16 17 18 19 1A	TD message (CF)
:	:	:	:	:	:	:
0.0010	1	7E0	Request	8	23 F4 F5 F6 F7 F8 F9 FA	TD message (CF)
0.0009	1	7E0	Request	5	24 FB FC FD FE	TD message (CF)
0.0011	1	7E8	Response	3	02 76 02	TD message
1.9617	1	7DF	Request	3	02 3E 80	TP message

Table 87 — TransferData – Module #4 (block #3)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.0007	1	7E0	Request	8	07 36 03 02 03 04 05 06	TD message (FF)
0.0011	1	7E8	Response	3	02 76 03	TD message
0.9982	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message

Table 88 — RequestTransferExit - Module #4

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.0013	1	7E0	Request	2	01 37	RTE message
0.0011	1	7E8	Response	2	01 77	RTE message
1.9976	1	7DF	Request	3	02 3E 80	TP message

Table 89 — RoutineControl(check programming dependencies)

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
1.0012	1	7E0	Request	5	04 31 01 FF 01	RC message
0.0001	1	7E8	Response	4	03 7F 31 78	NR w/ NRC78
0.9987	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message
0.0011	1	7E8	Response	4	03 7F 31 78	NR w/ NRC78
1.9990	1	7DF	Request	3	02 3E 80	TP message
1.0019	1	7E8	Response	5	04 71 01 FF 01	RC message
0.9982	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message

Table 90 — WriteDataByIdentifier - recordDataIdentifier = VIN

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.0004	1	7E0	Request	8	10 14 2E F1 90 57 41 4C	WDBI msg. (FF)
0.0001	1	7E8	Response	3	30 00 00	FlowControl
0.0012	1	7E0	Request	8	21 54 4F 4E 53 2D 57 45	WDBI msg. (CF)
0.0010	1	7E0	Request	8	22 42 2E 43 4F 4D 20 20	WDBI msg. (CF)
0.0011	1	7E8	Response	4	03 6E F1 90	WDBI message
1.9961	1	7DF	Request	3	02 3E 80	TP message
2.0001	1	7DF	Request	3	02 3E 80	TP message

9.4.3 Programming phase #1 - Post-Programming step

Table 91 — ECUReset - hardReset

Relative Time	Channel #	CAN ID	Client Request / Server Response	DLC	PCI and frame data bytes	Comments
0.3946	1	7DF	Request	3	02 11 01	ER message
0.0011	1	7E8	Response	3	02 51 01	ER message
0.0001	1	7E9	Response	3	02 51 01	ER message

Annex A
(normative)

Network configuration data recordDataIdentifier definitions

A.1 Network configuration data recordDataIdentifier definitions

The network configuration data related recordDataIdentifiers that can be read via the ReadDataByIdentifier (22 hex) service can be used by an external test equipment or any other client to read out information about the communication protocol and CAN network configuration supported by the vehicle communication system. The client can request information about:

— the addressing scheme that shall be used,

— the CAN identifiers that shall be used by the client and the server(s) for the subsequent communication,

— the servers that are connected to a remote network.

The network configuration data allows the client to dynamically set-up the communication configuration and adopt to the addressing format and parameters provided by the vehicle system server(s).

The network configuration data will be requested via the ReadDataByIdentifier (22 hex) service using either physical and functional addressing.

The recordDataIdentifiers defined below can be supported by any server. In most cases, the server is the Diagnostic Configuration Master. A system may have a Diagnostic Configuration Master, but it is also possible to let individual servers directly report their own data.

The following table defines the recordDataIdentifier values applicable for network configuration data (reference ISO 14229-1 for the reserved range of recordDataIdentifiers for network configuration data).

Table A.1 — recordDataIdentifier definitions for network configuration data

Hex	Description	Cvt	Mnemonic
F010	**addressFormat** This value shall cause the Diagnostic Configuration Master server to report which addressing scheme shall be used for the subsequent communication.	U	AF
F011	**responseCANId** This value shall cause the Diagnostic Configuration Master server, or any other server, to report a list of CAN Ids that will be used by vehicle servers for responses during the subsequent communication. The list shall contain all response CAN Ids that are associated with the target address included in the request message to the Diagnostic Configuration Server (or any other server). This means that if a functional target address is used for the request, then the response shall contain a list of all CAN Ids that will be used by servers supporting that functional target address. If a physical target address is used for the request, then the response shall contain the CAN Id that will be used for responses from the server having that physical target address.	U	RSPCID

Table A.1 — recordDataIdentifier definitions for network configuration data

Hex	Description	Cvt	Mnemonic
F012	**physicalRequestCANId** This value shall cause the Diagnostic Configuration Master server, or any other server/servers, to report a list of all CAN Ids that shall be used by the client to send physically addressed request messages during the subsequent communication. The response is depending on whether the request is sent with a functional or physical target address. If a functional target address is used for the request, the response shall either contain a list with all CAN Ids that shall be used by the client for physically addressed requests to any server in the vehicle system (in this case shall only one server respond), or each responding server shall send a list with one CAN Id that shall be used for physically addressed requests to that server. NOTE: For vehicle systems using extended addressing there is always only one CAN Id in this list. If a physical target address is used for the request, the response shall contain the CAN Id that shall be used for requests to the server having that physical target address.	U	PRQCID
F013	**functionalRequestCANId** This value shall cause the Diagnostic Configuration Master server, or any other server, to report the CAN Id that shall be used during the subsequent communication, by the client, for functionally addressed request messages, corresponding to the functional address included in the request message. NOTE: For vehicle systems using extended addressing the same CAN Id is always used for all functionally addressed request messages.	U	FRQCID
F014	**allFunctionalRequestCANIds** This value shall cause the Diagnostic Configuration Master server, or any other server, to report a list of all CAN Ids that shall be used by the client, to send functionally addressed request messages, during the subsequent communication. NOTE: For vehicle systems using extended addressing there is always only one CAN Id in this list	U	AFRQCID
F015	**remoteServerAddress** This value shall cause the Remote Diagnostic Configuration Master server, or any other server, to report a list of remote server identifiers (server target/source addresses) for servers connected to a remote network. The response is depending on whether the request is sent with a functional or physical remote target address. If a functional remote target address is used for the request, then each remote server shall send a response message with its own remote server identifier. If a physical remote target address is used for the request, then the Remote Diagnostic Configuration Master server, or any other server, shall sent a list including the remote server identifiers for all remote servers on the remote network.	U	RSA
F016	**serverDiagnosticAddress** This value shall cause the server(s) to report its (their) physical diagnostic address(es). Independent of the addressing method used in the request message the response always contain the single diagnostic address (target address) of the requested server(s)..	U	SDA

A.2 Network configuration data recordDataIdentifier data format definitions

The following sections define the data parameter format and the content of the network configuration recordDataIdentifiers as given in the table above.

Table A.2 — addressFormat recordDataIdentifier data format
recordDataIdentifier = addressFormat

Byte pos.	Parameter Name	Cvt	Hex Value	Mnemonic
#1	dataRecord[] = [serverAddressFormat]	M	00-FF	DREC_ DATA_1

Table A.3 — CAN Id related recordDataIdentifier's data formats
recordDataIdentifier = physicalRequestCANId, functionalRequestCANId, allFunctionalRequestCANIds

Byte pos.	Parameter Name	Cvt	Hex Value	Mnemonic
	dataRecord[] = [			DREC_
#1	diagnosticAddress #1	M	00-FF	DATA_1
#2	CANIdBits24to28 #1	M	00-FF	DATA_2
#3	CANIdBits16to23 #1	M	00-FF	DATA_3
#4	CANIdBits8to15 #1	M	00-FF	DATA_4
#5	CANIdBits0to7 #1	M	00-FF	DATA_5
:	:	:	:	:
#n-4	diagnosticAddress #m	C	00-FF	DATA_(n-4)
#n-3	CANIdBits24to28 #m	C	00-FF	DATA_(n-3)
#n-2	CANIdBits16to23 #m	C	00-FF	DATA_(n-2)
#n-1	CANIdBits8to15 #m	C	00-FF	DATA_(n-1)
#n	CANIdBits0to7 #m]	C	00-FF	DATA_n

C: The Diagnostic Configuration Master might report multiple diagnosticAddress and CAN Ids sets, depending on the target address used in the request message that causes the diagnostic configuration master to respond (see network configuration data recordDataIdentifier definitions for more details).

Table A.4 — Diagnostic address related recordDataIdentifier's data format
recordDataIdentifier = diagnosticAddress

Byte pos.	Parameter Name	Cvt	Hex Value	Mnemonic
#1	dataRecord[] = [diagnosticAddress]	M	00-FF	DREC_ DATA_1

Table A.5 — addressFormat recordDataIdentifier data format
recordDataIdentifier = remoteServerAddress

Byte pos.	Parameter Name	Cvt	Hex Value	Mnemonic
	dataRecord[] = [			DREC_
#1	remoteServerAddress #1	M	00-FF	DATA_1
:	:	:	:	:
#n	remoteServerAddress #m]	C	00-FF	DATA_n

C: In case there are multiple remote servers then this parameter is present. The format of the remoteServerAddress is identical to the format of the diagnosticAddress parameter.

The following tables define the data parameter values of the recordDataIdentifiers defined above.

Table A.6 — serverAddressFormat data parameter value definition

Hex	Description	Cvt	Mnemonic
01	**normalAddressingFormat** The server(s) support(s) a network layer protocol as specified in ISO 15765-2 using the normal addressing scheme.		
02	**fixedNormalAddressingFormat** The server(s) support(s) a network layer protocol as specified in ISO 15765-2 using the fixed normal addressing scheme.		
03	**extendedAddressingFormat** The server(s) support(s) a network layer protocol as specified in ISO 15765-2 using the extended addressing scheme.		
04	**mixedAddressingFormat** The server(s) support(s) a network layer protocol as specified in ISO 15765-2 using the mixed addressing scheme.		

Table A.7 — diagnosticAddres / remoteServerAddress data parameter value definition

Hex	Description	Cvt	Mnemonic
00 - FF	**diagnosticAddress / remoteServerAddress** This is the diagnostic address for the server associated with the CAN identifier being reported. It is the physical address of the server, if the recordDataIdentifier equals responseCANId or physicalRequestCANId. It is the functional address of the server(s), if the recordDataIdentifier equals functionalRequestCANId or allFunctionalRequestCANId. It is the physical address of the server(s) addressed if the recordDataIdentifier equals serverDiagnosticAddress.		DA

Table A.8 — CANIdBits24to28 data parameter value definition

Hex	Description	Cvt	Mnemonic
00 - 1F	**29 bit CAN Identifiers** This is the value of bits 24-28 in the 29-bit CAN identifier, positioned with identifier bit 24 as the least significant bit. The three most significant bits of this parameter shall always be set to 0 (zero) for 29-bit CAN identifiers.		29BCID2428
FF	**11 bit CAN Identifiers** This value is used to indicate that the parameters CANIdBits8to15 and CANIdBits0to7 contains an 11-bit CAN identifier.		11BCID2428

Table A.9 — CANIdBits16to23 data parameter value definition

Hex	Description	Cvt	Mnemonic
00 - FF	**11 bit CAN Identifiers / 29 bit CAN Identifiers** For 11-bit CAN identifiers this parameter shall be treated as don't care. For 29 bit CAN identifiers this is the value of bits 16-23 in the 29-bit CAN identifier, positioned with identifier bit 16 as the least significant bit.		CID1623

Table A.10 — CANIdBits8to15 data parameter value definition

Hex	Description	Cvt	Mnemonic
00 - FF	**11 bit CAN Identifiers / 29 bit CAN Identifiers** For 11 bit CAN identifiers this is the value of bits 8-10 in the 11-bit CAN identifier, positioned with identifier bit 8 as the least significant bit. The five most significant bits of this parameter shall be treated as don't care. For 29 bit CAN identifiers this is the value of bits 8-15 in the 29-bit CAN identifier, positioned with identifier bit 8 as the least significant bit.		CID0815

Table A.11 — CANIdBits0to7 data parameter value definition

Hex	Description	Cvt	Mnemonic
00 - FF	**11 bit CAN Identifiers / 29 bit CAN Identifiers** For 11 bit and 29 bit CAN identifiers this is the value of bits 0-7 in the CAN identifier, positioned with identifier bit 0 as the least significant bit.		CID0007