CANdesc

Technical Reference

Version 2.19.00

Authors:	Oliver Garnatz, Mishel Shishmanyan, Stefan Hübner, Matthias Heil
Version:	2.19.00
Status:	released (in preparation/completed/inspected/released)

1 History

Author	Date	Version	Remarks
Oliver Garnatz	2003-11-12	2.00.00	Splitting into separate documents and general revision
Oliver Garnatz	2004-01-13	2.00.01	Added chapter ‘Application interface flow’ Updated format template
Mishel Shishmanyan	2004-03-09	2.01.00	New application callback convention (from CANdesc 2.09.00)
Mishel Shishmanyan	2004-03-29	2.02.00	New APIs: - **DescGetActivityState** (from CANdesc 2.10.00) - **DescSchedulerTask()** (from CANdesc 2.09.00)
Mishel Shishmanyan	2004-04-26	2.03.00	Added more information and limitations about the ring-buffer mechanism (6.6.8 “Ring Buffer Mechanism”) New feature: - **Support for generic user service** (from CANdesc 2.11.00) - **Force CANdesc to send RCR-RP response** (from CANdesc 2.11.00)
Stefan Hübner	2004-07-16	2.03.01	Editorial revision
Oliver Garnatz	2004-08-12	2.04.00	Added chapter 4.2 ReadDataByIdentifier (SID $22) within the Single- and the Multiple PID mode is described
Oliver Garnatz	2004-10-08	2.05.00	ESCAN0000982: Description of MainHandler structure is not readable ROE transmission unit is described in detail
Stefan Hübner Oliver Garnatz	2004-10-15	2.06.00	Some additional information are provided
Peter Herrmann Klaus Emmert	2005-06-22	2.07.00	**Added:** Service $2C description. **Added:** Warning Text added
Mishel Shishmanyan Oliver Garnatz	2005-08.03	2.08.00	**API added:** - DescStateTask, - DescTimerTask,

			- DescMayCallStateTaskAgain. - ApplDescFatalError **API modified:** - DescTask, - ApplDescCheckSessionTransition, - DescGetActivityState, - DescGetStateSession. **API removed:** - DescSchedulerTask Modified description for ReadDataByIdentifier with long data and negative response in main-handler.
Oliver Garnatz	2006-03-02	2.09.00	**Added:** ...prevent the ECU going to sleep while diagnostic is active
Mishel Shishmanyan	2006-03-24	2.10.00	**Added:** document overview
Mishel Shishmanyan	2006-04-27	2.11.00	**Modified:** -6.6.12 DynamicallyDefineDataIdentifier ($2C) (UDS) functions -6.6.12.1 DescMayCallStateTaskAgain()
Mishel Shishmanyan	2007-02-22	2.12.00	**Added:** *- 6.6.8.3 “DescRingBufferCancel()”*
Matthias Heil	2008-01-03	2.13.00	**Added:** **Caution** concerning user main handler on protocol level
Matthias Heil	2008-02-29	2.14.00	**Added:** Handling of read/write memory by address: *- 5.5 “Read/Write Memory by Address”* *- 6.6.7.2 “DescStartMemByAddrRepeatedCall()”* *- 6.6.13 ”Memory Access Callbacks”*
Mishel Shishmanyan	2008-06-06	2.15.00	***Removed:*** *Chapter “ResponseOnEvent Transmission Unit”* ***Added:***

			- 6.6.12.3 “Non-volatile memory support”
Mishel Shishmanyan	2008-11-09	2.16.00	***Modified:*** *- 6.6.8* and *6.6.8.1*: Added limitation for UDS and SPRMIB with the ring buffer usage. *- 7.6 …work with the ring-buffer mechanism* ***Added:*** *- 6.6.14 Flash Boot Loader Support* *- 7.8 …send a positive response without request after FBL flash job*
Mishel Shishmanyan	2009-05-18	2.17.00	***Modified:*** *6.6.5.1ApplDescCheckSessionTransition()* ***Added:*** *6.6.5.3DescIsSuppressPosResBitSet ()*
Mishel Shishmanyan	2009-08-11	2.18.00	***Modified:*** Minor editorial changes *5.2 Configure Handlers using CANdela attributes* – added new data object attributes ***Added:*** *7.9 …enforce CANdesc to use ANSI C instead of hardware optimized bit type* *5.1 Configure DBC attributes for diagnostics*
Mishel Shishmanyan	2010-12-21	2.19.00	***Modified:*** *6.6.8.2 DescRingBufferWrite()* *6.6.13.1 ApplDescReadMemoryByAddress()* *6.6.13.2 ApplDescWriteMemoryByAddress()*

Contents

Illustrations

2 Introduction

This document has not the job to describe the diagnostic itself. The focus of this document is the technical aspects of the CANdesc component.

Please note
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector's release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

3 Documents this one refers to…

- User Manuals CANdesc and CANdescBasic (one for both)
- Docu OEM

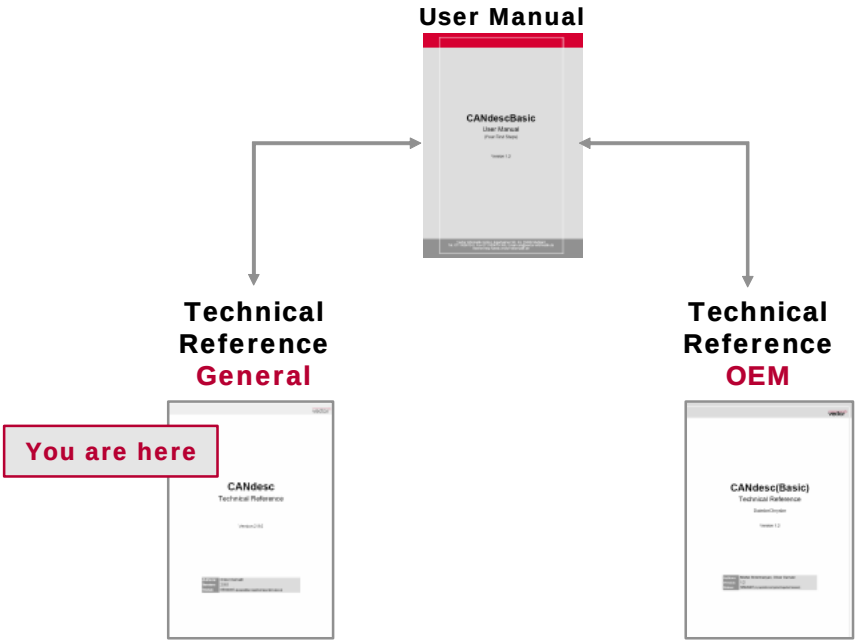


Figure 3-1: Manuals and References for CANdesc

All common topics with CANdesc and CANdescBasic are described within this technical reference very detailed.

Read all about OEM-specific differences in the TechnicalReference_OEM.

For faster integration, refer to the product's corresponding user manual CANdesc or CANdescBasic.

4 Architecture Overview

This chapter should describe the internal structure and behavior of the CANdesc component.

4.1 CANdesc – Internal processing

4.1.1 Diagnostic protocol

The communication described in the diagnostic protocol consists of a ping-pong communication between a tester (client) and an ECU (server). The tester requests a service in the ECU by transmitting a request to him. The ECU should response with a positive response, if the result of this service is valid or the action is prepared to be done. Is the result negative or the action could not be executed, the ECU should respond negative.

The validity checks have typically the same pattern for all services (as shown in Figure 4-1: General request flow). These components which are included in this flow, build up the main base of the CANdesc component.

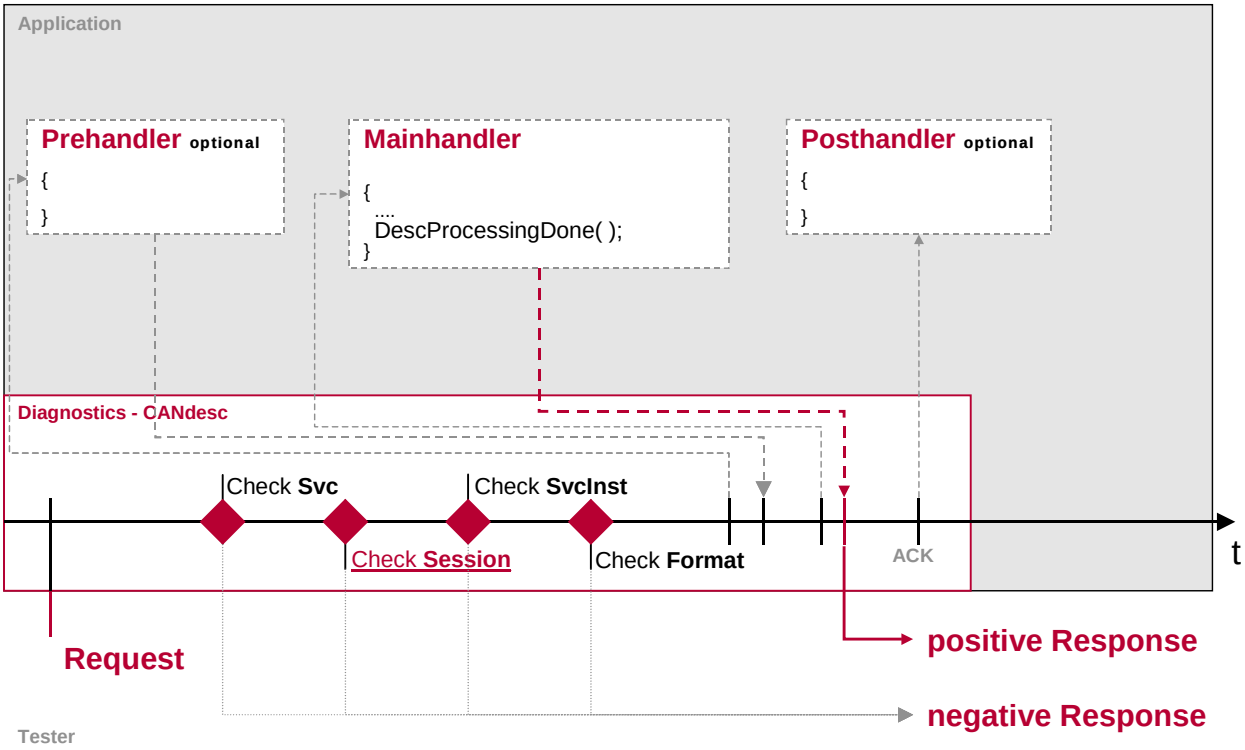


Figure 4-1: General request flow

4.1.2 How does this flow actually work?

The picture below shows a simply structured description of the module functionality.

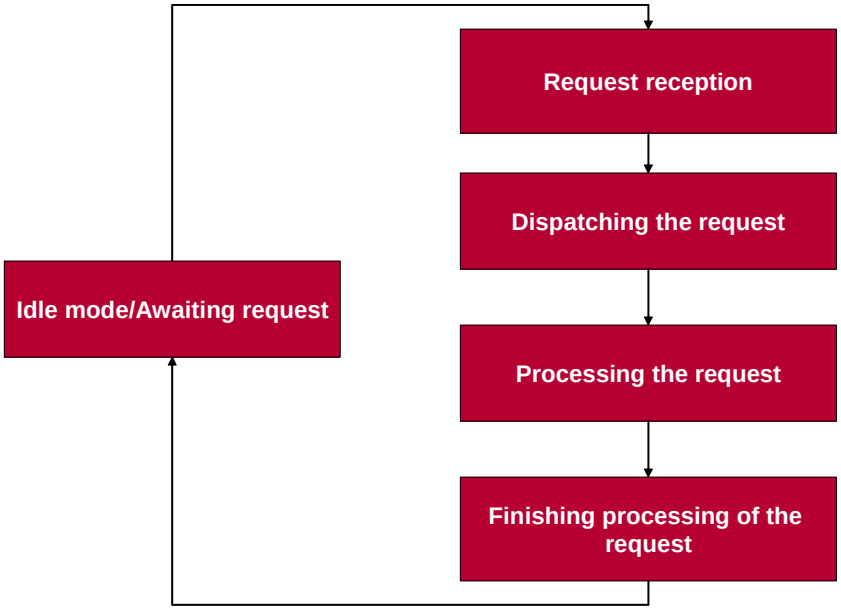


Figure 4-2: DESC run diagram

Lets assume that the component is currently in the **“Awaiting request”** state. In this state it waits for the next diagnostic request and if it is needed – it provides also timing monitoring.

Once a diagnostic request transmission was initiated from the transport layer, the component enters in the state **“Request reception”**. If the reception is finished, further physical requests will be blocked until the response is sent. Depending on the used OEM a functional request in the ISO 14230 standard will be handled parallel[1] to physical request. The ISO 14229-1 standard is more restricted to the parallel handling. Except the TesterPresent Service no other service could be handled parallel.

[1] Not all services could be handled parallel.

After the reception of the request is completed the request processing will be prepared. The component is in the **“Dispatching request”** state. The processing of the request is done at a task level within the next call of the DescTask() function.

First the SID is checked whether supported or not. If not a negative response ‘ServiceNotSupported’ (NRC $11) will be sent.

Next step is to check if the supported SID is permitted in the current Session (Diagnostic Mode). If not, the negative response ‘ServiceNotSupportedInTheCurrentSession’ (NRC $7F) is sent automatically by the CANdesc component.

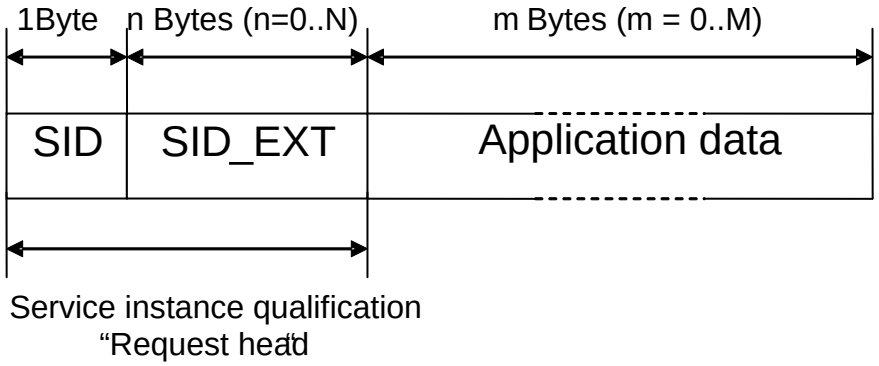


Figure 4-3: Request message mapping

After that the CANdesc component validates, if the sub-service (service instance) is supported or not. This is implemented with a powerful binary search. If the service instance is not supported, the request will be rejected with the corresponding error code ‘SubFunctionNotSupported’ (NRC $11, for service which have SubFunctions) or ‘InvalidFormat’ (NRC $13, for service with data identifiers).

For each service instance which is supported by the current configuration, the CANdesc component knows the exact length of most requests. (Some requests use variable data length elements thus a fixed length doesn’t exist.) If the length is known and it does not match, the dispatcher will reject this request (dependent to the manufacturer specification). If the complete request length is not known, the application has to do this job.

If the service instance is found, the state checks (e.g. ‘Security Level’) will be performed. If all of them are passed then the component enters the state **“Processing request”** in the diagram above. This state consists of several parts that are represented in more detailed structure shown below. The dotted lines reveal the optional parts for the implementation. For example – the Pre-, Post- and SignalHandlers are optional and might not be implemented.

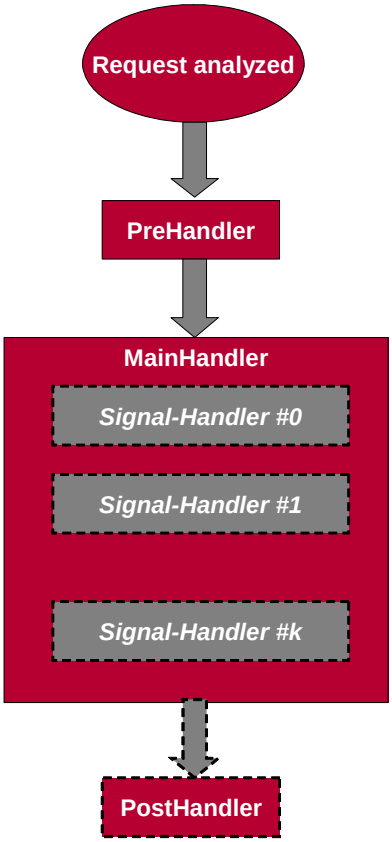


Figure 4-4: Request processing stages

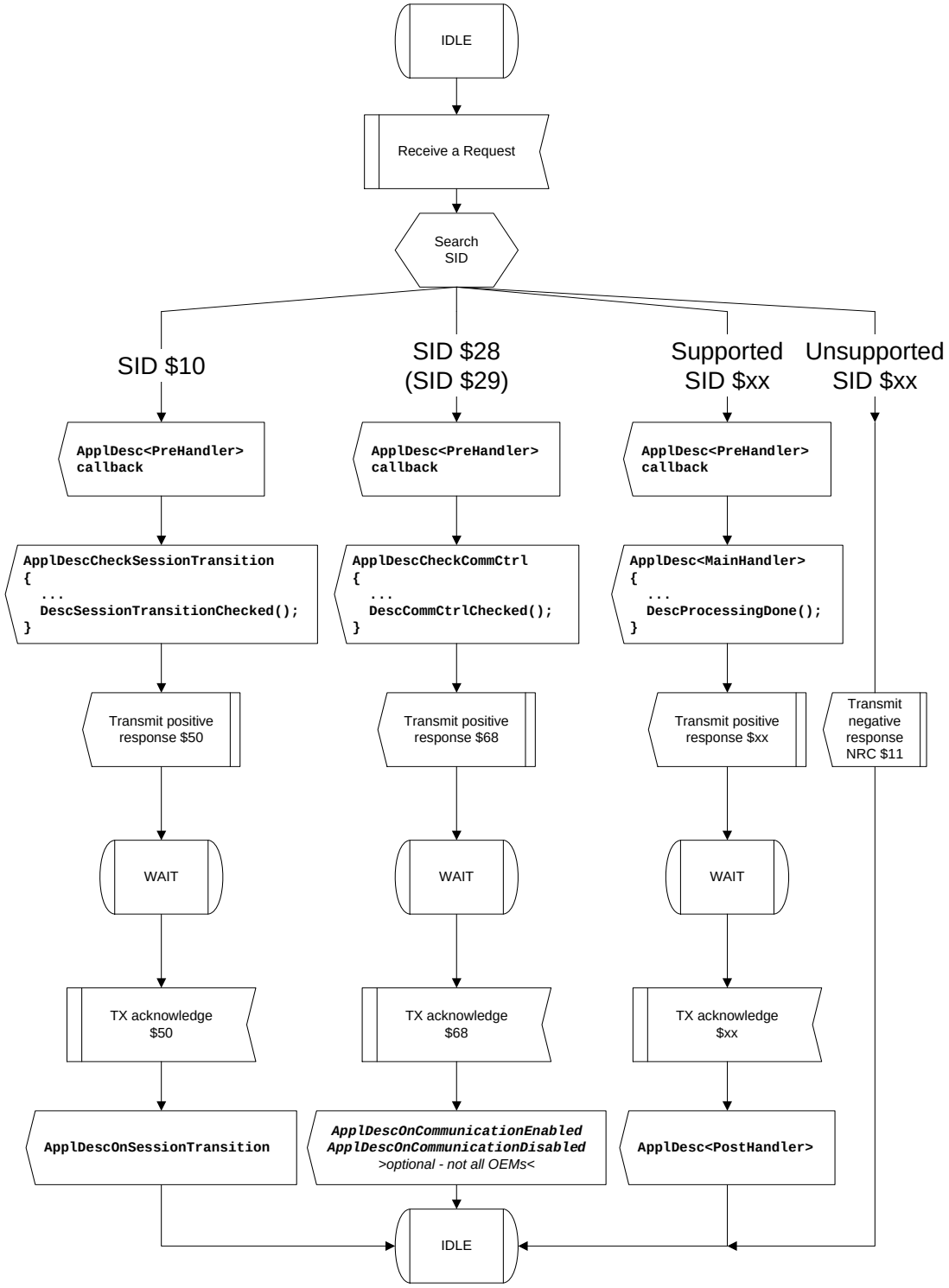
After the response is composed CANdesc must be informed about, to start the transmission of the final response. CANdesc is doing the handshake with the Tester (automatic transmission of RCR-RP) while the state “” is doing.

Within the end of the transmission the state “**Finishing processing of the request”** is entered and the PostHandler (if configured) is called. In this PostHandler the application has to do the closing (e.g. updating a state machine, prepare the ECU for a reset …). The session state for example (which is managed by CANdesc) is also updated in a PostHandler.

4.2 Application interface flow

4.2.1 Session- and CommunicationControl

The services SessionControl and CommunicationControl are typically handled by CANdesc. But the application still has the possibility to reject these service requests. You can find a detailed description in chapter 6.6.5 Session Handling and in chapter 6.6.6 CommunicationControl Handling also.



5 Advanced Configuration

5.1 Configure DBC attributes for diagnostics

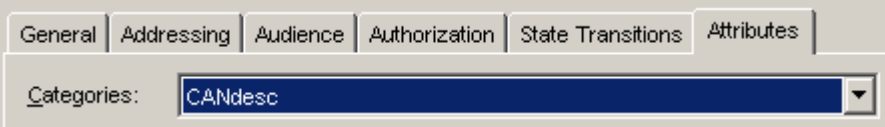
If the diagnostic messages shall be defined in the communication data-base file (DBC), and not received via CANdriver ranges (e.g. in case of normal fixed or extended addressing), the following attributes in the DBC file must exist and shall be set as shown below.

Attribute Name	Object Type	Value Type	Values the default value is written in bold	Description
DiagRequest	Message	Enum	**No** Yes	Specifies (Yes) that the message is a diagnostic **physical USDT** request message.
DiagResponse	Message	Enum	**No** Yes	Specifies (Yes) that the message is a diagnostic **USDT** response message.
DiagState	Message	Enum	**No** Yes	Specifies (Yes) that the message is a diagnostic **functional USDT** request message.
DiagUudtResponse	Message	Enum	**false** true	Specifies (true) that the message is a diagnostic **UUDT** response message.

Table 5-1: DBC file diagnostic message attributes

5.2 Configure Handlers using CANdela attributes

The following attributes are relevant for configuration of CANdela software components:



All attributes have the category ‘CANdesc’.

Caution
Please note: if you have a CDD file where one of the below listed attributes is already set to “oem”, you are not allowed to change this attribute value, since this will affect the proper CANdesc functionality for the OEM.

Diagnostic Class attributes	
MainHandlerSupport (on Protocol Service Level)	Type: enum: none = 0, oem = 1, user = 2 Provide MainHandlers and select type for all Protocol Services of Diagnostic class.
PreHandlerSupport (for all Protocol Services)	Type: enum: none = 0; oem = 1; user = 2

	Provide type of Service PreHandlers for all Protocol Services of Diagnostic class. Restriction: Only evaluated if 'MainHandlerSupport (on Protocol Service Level)' is set, i.e. unequals to 'none'.
PostHandlerSupport (for all Protocol Services)	Type: enum: none = 0; oem =1; user = 2 Provide type of Service PostHandlers for all Protocol Services of Diagnostic class. Restriction: Only evaluated if 'MainHandlerSupport (on Protocol Service Level)' is set, i.e. unequals to 'none'.
PacketHandlerSupport	Type: enum: none = 0; all = 1 Switch on/off Packet Handlers support for each Diagnostic instance of current Diagnostic class. Restriction: Only evaluated if 'MainHandlerSupport (on Protocol Service Level)' is set, i.e. unequals to 'none'.

Table 5-2: Names of CANdela Diagnostic Class attributes

Caution

In some cases, the CANdesc service implementations of one service requires subservice handling of other services. Which services - if any at all - are affected by such a dependency is manufacturer specific. Thus, the attribute 'MainHandler-Support (on Protocol Service Level)' must be used carefully.

Before disabling the subservice handling (by setting the attribute to 'user'), check the manufacturer specific documentation. If you are in doubt, please contact Vector to verify CANdesc will work correctly with your settings.

Diagnostic Instance attributes	
PacketHandlerOption	Type: enum: *user*= 0; generated= 1 Provide type of current Packet Handler for first found Service Positive response. Restriction: Only evaluated if Folder attribute 'PacketHandlerSupport' is set to 'all'.

Table 5-3: Names of CANdela Diagnostic Instance attributes

Service attributes	
MainHandlerSupport (on Service Level)	Type: enum: *user* = 0; oem = 1; generated = 2 Provide MainHandler for current Service (Service/Subfunction combination). Restriction: Only evaluated if Folder attribute 'MainHandlerSupport (on Protocol Service Level)' is not set, i.e. equals to 'none'.
PreHandlerSupport	Type: enum: *none* = 0; oem = 1; user = 2 Provide type of Service PreHandler for current Service (Service/Subfunction combination). The Service PreHandler is executed directly before the MainHandler. Restriction: Only evaluated if Folder attribute

	‘MainHandlerSupport (on Protocol Service Level)’ is not set, i.e. equals to ‘none’.
PreHandlerOverrideName	Type: string: {empty} Provide alternative name for Service PreHandler instead of generated one (default prefix is added to PreHandlerOverrideName).You can reuse one Service PreHandler for different services, if you specify the same Override Name. Restriction: Only evaluated if Folder attribute ‘MainHandlerSupport (on Protocol Service Level)’ is not set, i.e. equals to ‘none’.
PostHandlerSupport	Type: enum: *none* = 0; oem = 1; user = 2 Provide type of Service PostHandler for current Service (Service/Subfunction combination). The Service PostHandler is executed after a positive confirmation of the positive response. Restriction: Only evaluated if Folder attribute ‘MainHandlerSupport (on Protocol Service Level)’ is not set, i.e. equals to ‘none’.
PostHandlerOverrideName	Type: string: {empty} Provide alternative name for Service PostHandler instead of generated one (default prefix is added to PostHandlerOverrideName).You can reuse one Service PostHandler for different Services, if you specify the same Override Name. Restriction: Only evaluated if Folder attribute ‘MainHandlerSupport (on Protocol Service Level)’ is not set, i.e. equals to ‘none’.

Table 5-4: Names of CANdela Service attributes

Data Object attributes	
VariableForDirectAccess	Type: string: {empty} Specify name of application variable to be accessed by CANdesc. Currently, no (extended) type specifiers are allowed. If no variable for direct access is specified, a callback function will be generated.
SignalHandlerOverrideName	Type: string: {empty} Provide alternative name for Signal Handler instead of generated one (default prefix is added to SignalHandlerOverrideName). You can reuse one Signal Handler for different signals, if you specify the same Override Name. Pay attention that the function signature must match the different use-cases.
SignalPrototype	Type: enum: Generated = 0, GeneratedConst = 1, GeneratedUserDefined = 2, None = 3 Provides the information about the generation type: - "Generated" - RAM variables (default as we had up to now) - "GeneratedConst" - ROM variables - "GenratedUserDefined" - use own types (use the below attribute to determine the type (string)) - "None" - no prototype will be generated. Instead a header file will be included "DescType.h" where the user have to define his typedefs (for structure access for example). Restriction: Only evaluated if: - Service attribute ‘MainHandler (on Service Level)’ is set to ‘generated’ or Diagnostic instance attribute ‘PacketHandlerOption’ is set to ‘generated’. - ‘VariableForDirectAccess’ is set to refer an object.
UserDefinedQualifier	Type: string: {empty} Provides the information about the generation prototype name (e.g. instead using the Vector convention (vuint8, vuint16, etc. you can use uint8, t_StructType, etc.). Restriction: Only evaluated if: - Service attribute ‘MainHandler (on Service Level)’ is set to ‘generated’ or Diagnostic in-stance attribute ‘PacketHandlerOption’ is set to ‘generated’. - ‘VariableForDirectAccess’ is set to refer an object. - ‘SignalPrototype’ is set to ‘GenratedUserDefined’.

Table 5-5: Names of CANdela Data Object attributes

Please note that not it is not possible to combine all kinds of handlers inside of one Diagnostic instance (and sometimes, Diagnostic class).

The following flow chart shows the conditions and order in which the attributes are evaluated.

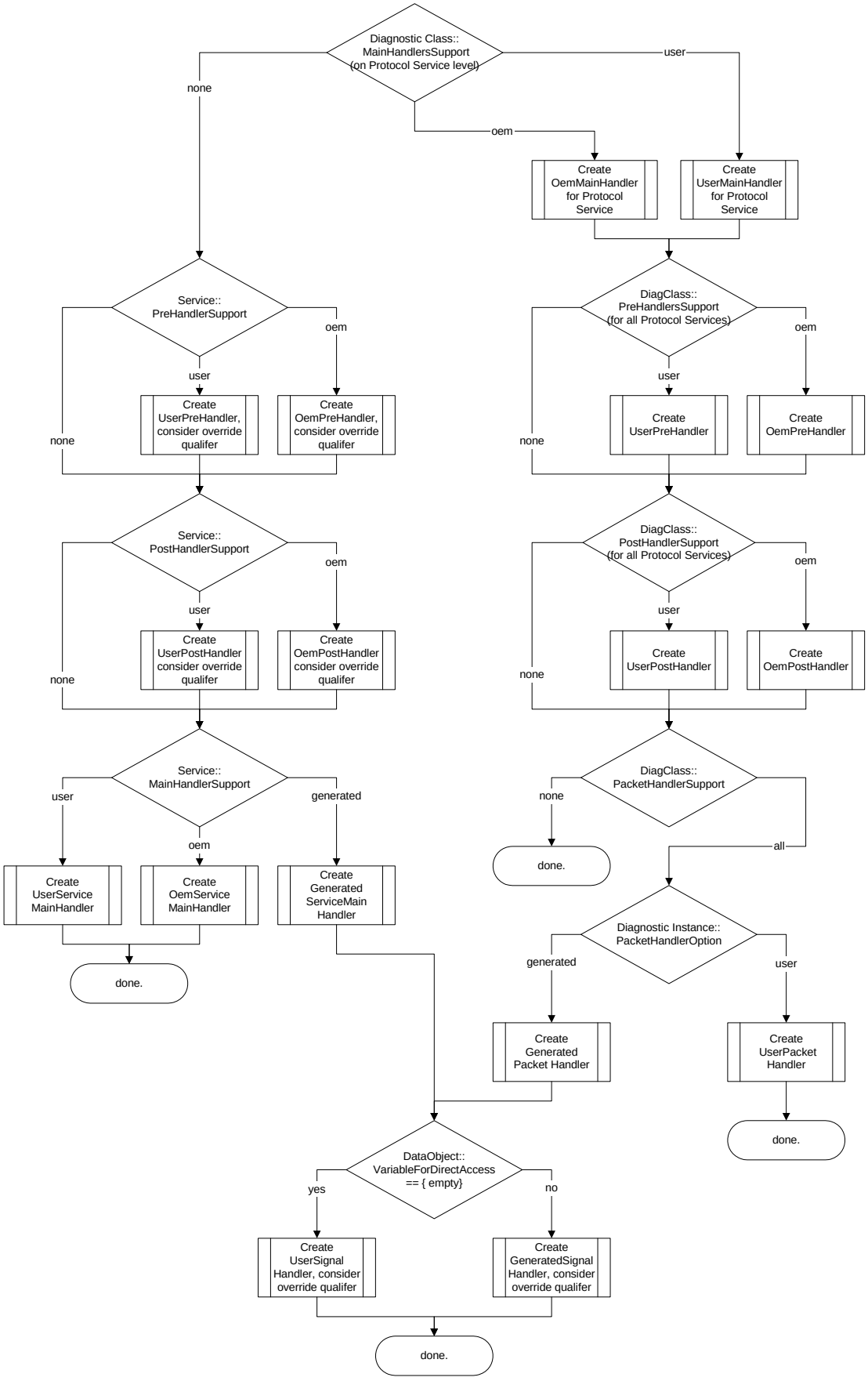


Figure 5-1: Dependency of CANdesc Handler configuration

5.3 ReadDataByIdentifier (SID $22)

This service has the purpose to read some predefined data records (PID). Each PID has a concrete data structure which is designed by CANdelaStudio.

As the standard case the request contains a single PID. This results in a single response containing the data structure of the record.

Single PID mode (well know case) example for PID $1234

Tester‘s request:

$22	$12	$34

ECU‘s response:

$62	$12	$34	Data block

The UDS allows to request multiple PIDs in a single request. This results is also a single response including the data structure of each requested PID.

Multiple PID mode example for PIDs: $1234, $ABCD

Tester‘s request:

$22	$12	$34	$AB	$CD

ECU‘s response:

$62	$12	$34	Data block	$AB	$CD	Data block

CANdesc will hide this multiple PID processing from the application. To do that some minor limitations in the interface has to be made (see chapter 5.3.2 Single PID mode). To show the differences, we discuss first the standard case. In the standard case there is no multiple PID processing possible. The second chapter (5.3.3 Multiple PID mode) is showing the multiple PID processing.

Which mode is used depends on the configuration (typically the OEM).

5.3.1 Limitations of the service

Session management

This service contains no sub-function identifier which means the global state group “session” may not be selected as a “relevant group” for any instance of this service. If there is a need for a PID to be rejected under a certain session, all PIDs must follow this rule and be specified to be rejected for this session. As a result the whole SID $22 will be rejected for this session. This behavior is harmonized with the UDS protocol specification, which allows service identifiers to be rejected in a session but no parameter identifiers.

5.3.2 Single PID mode

The Single PID mode is configured automatically, if the number of PIDs that can be requested at the same time, is limited to one PID. If more than one PID is requested, the request will be rejected with ‘RequestOutOfRange’ (NRC $31).

If the multiple PID mode of CANdesc is deactivated, the service $22 will be executed and processed like any other diagnostic service without any additional specifics or limitations.

5.3.2.1 Sending a positive response using linear buffer access

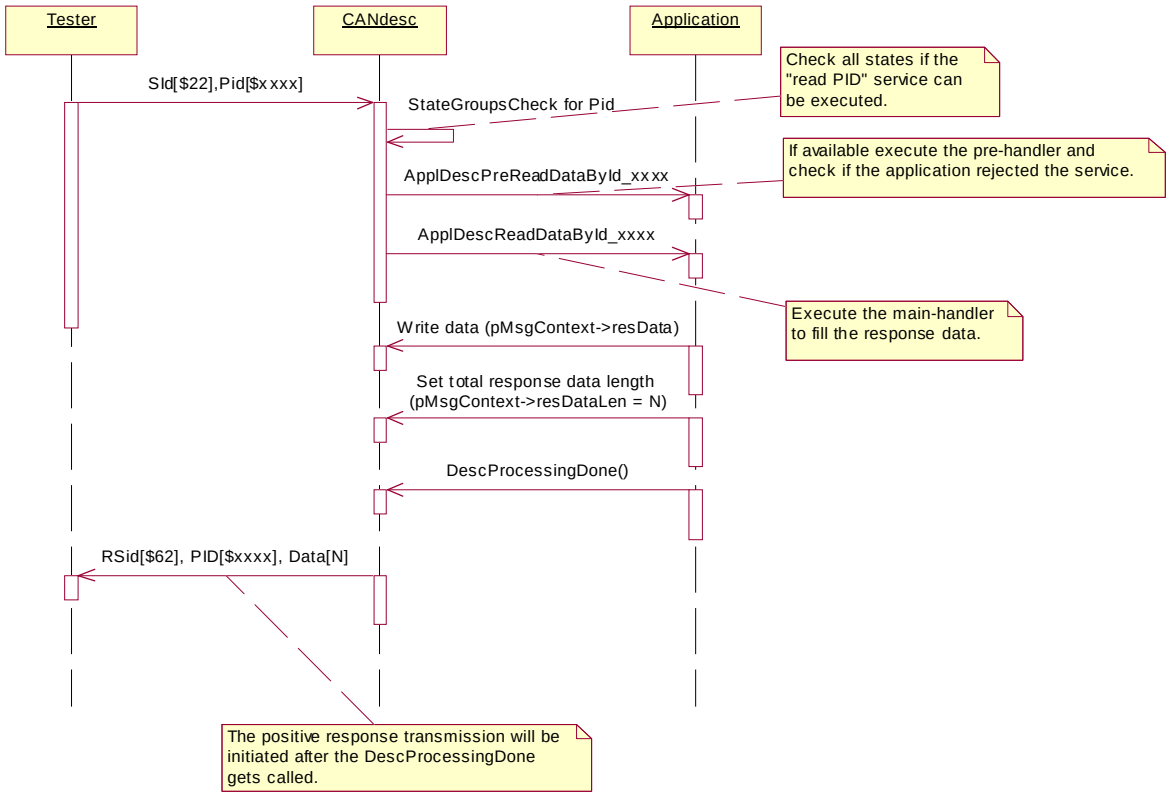


Figure 5-2: Linearly written positive response on single PID request

5.3.2.2 Sending a positive response using ring buffer access

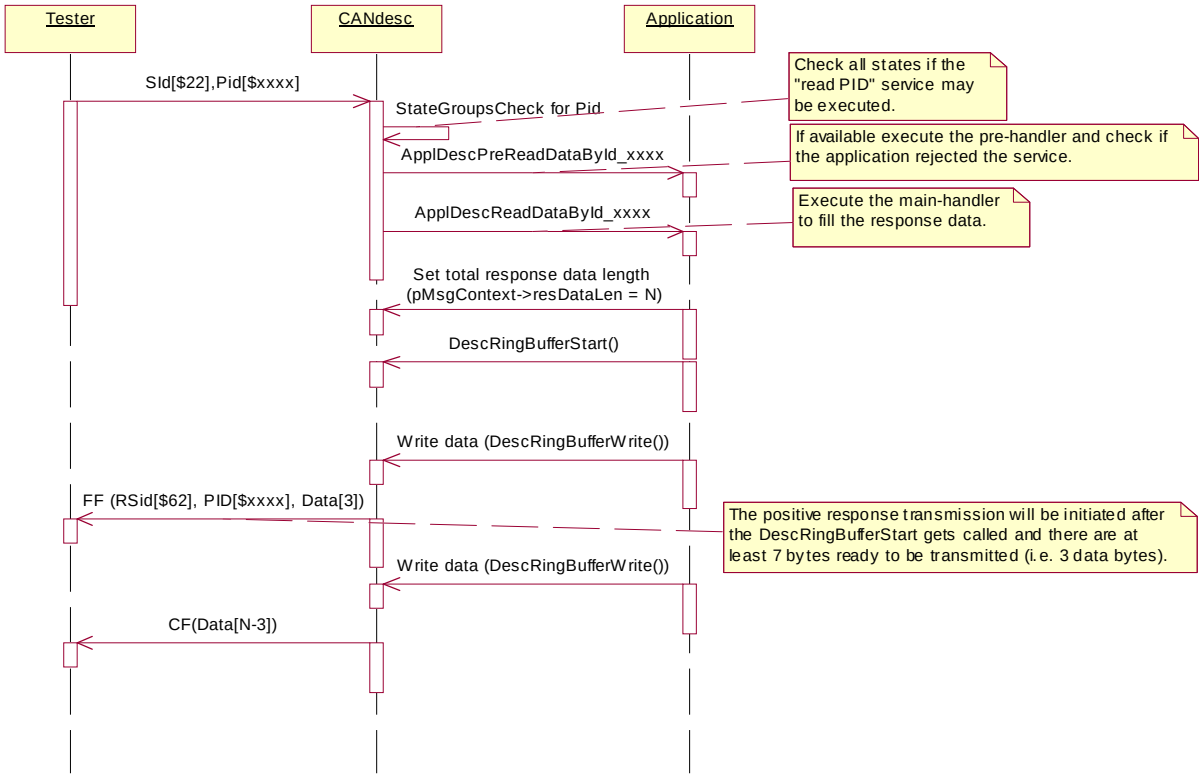


Figure 5-3: “On the fly” response data writing.

5.3.2.3 Sending a negative response

Due to the fact that the negative response handling has changed in the multiple PID mode, we recommend to do the same handling in the Single PID mode, too. Please refer the chapter 5.3.3.2 “Ring buffer active configuration” for the recommended negative response handling.

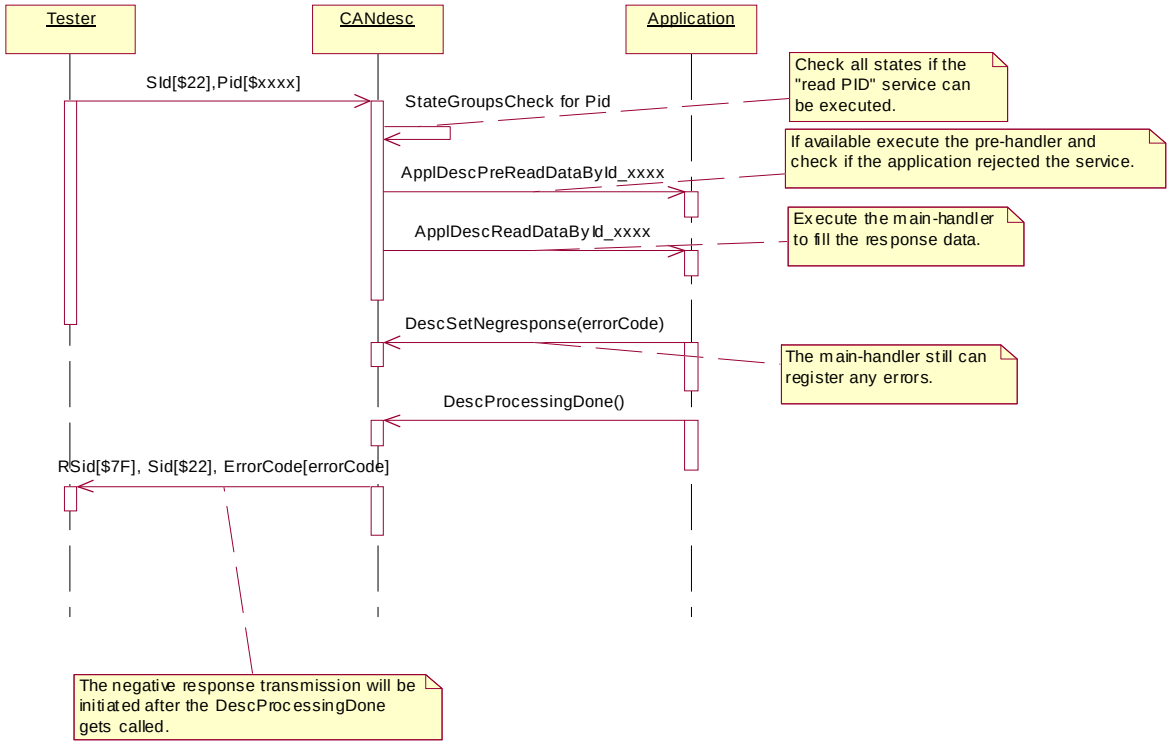


Figure 5-4: Negative response on single PID

5.3.3 Multiple PID mode

The Multiple PID mode is configured automatically if the number of PIDs, that can be requested at the same time, is greater than one. If more than this predetermined number of PIDs is requested, the request will be rejected with ‘RequestOutOfRange’ (NRC $31).

In this configuration some minor limitations must be taken into account while using the CANdesc interfaces.

For the service “ReadDataByIdentifier” the ring-buffer feature can be used. Depending on the usage of this feature, there are two main use cases for the multiple PID mode.:

5.3.3.1 Pure linear buffer configuration

The ring-buffer feature is deactivated in general.

If the system doesn't use any ring buffer access for filling the response, the PID pipeline is still quite simple and therefore with less limitations to the CANdesc API usage and application performance.

5.3.3.1.1 Sending a positive response

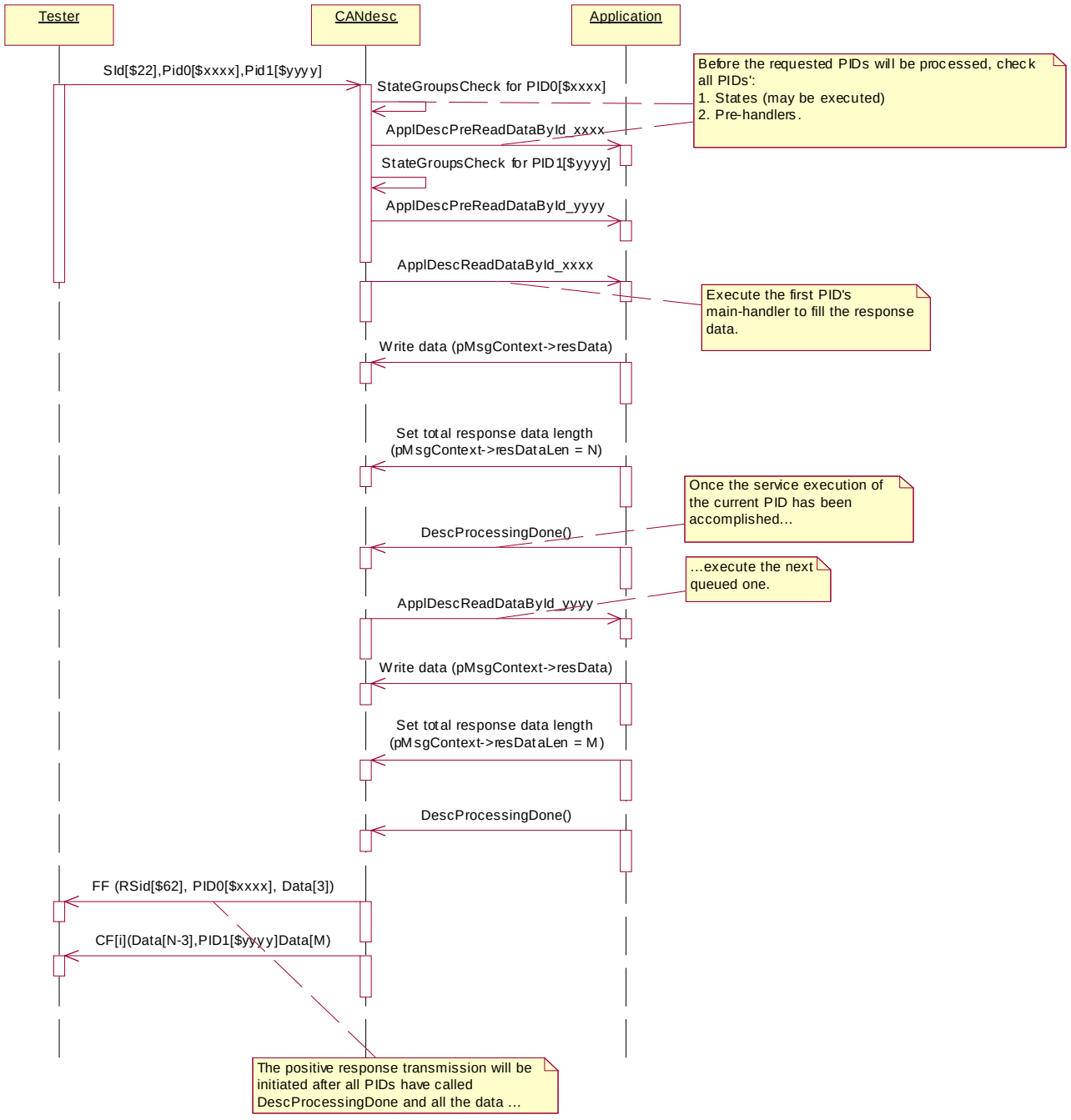


Figure 5-5: Linearly written positive response on multiple PIDs (global ring buffer option is off)

5.3.3.1.2 Sending a negative response

This example depicts the case where from two requested PIDs the first one may not be accessible and rejects the service execution.

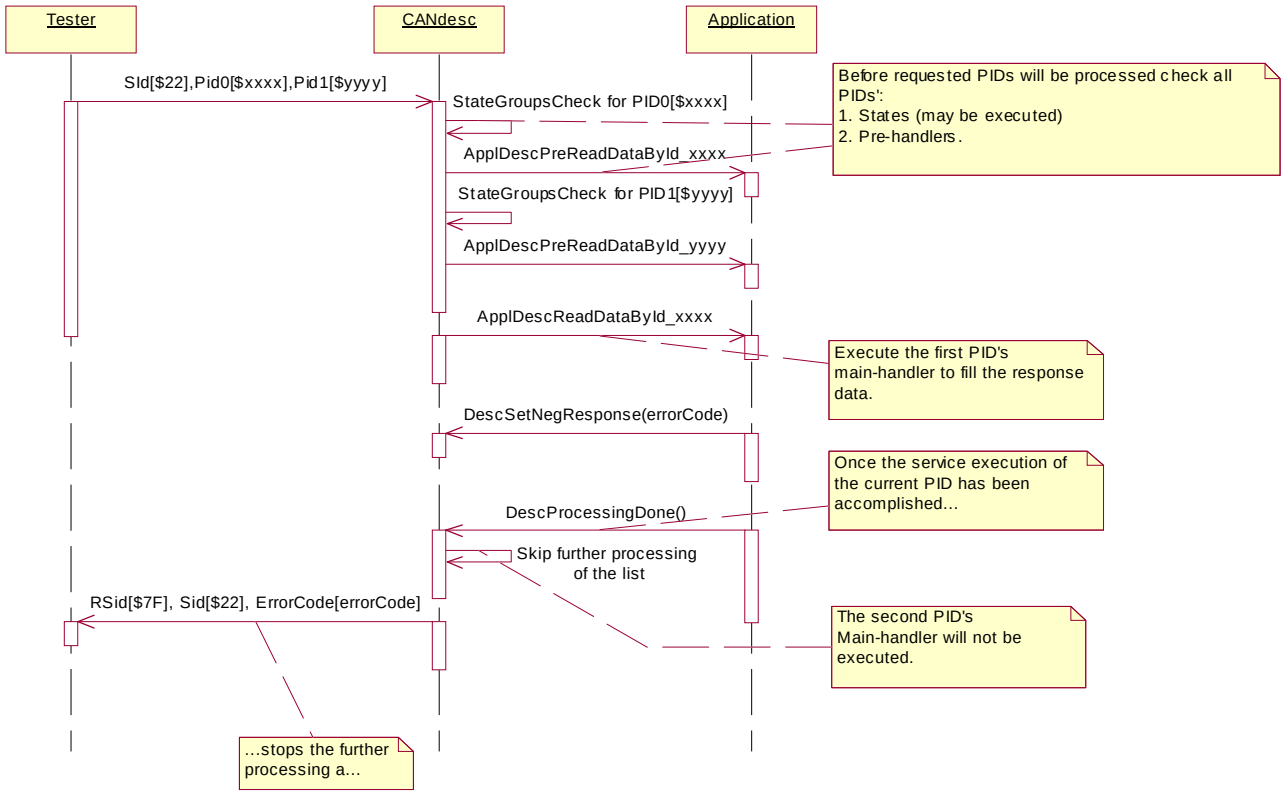


Figure 5-6: Negative response on multiple PIDs (global ring buffer option is off)

5.3.3.2 Ring buffer active configuration

Attention: The Ring-Buffer in 'Multiple PID' services can be first-time used since CANdesc version 2.13.00

Different concepts for the buffer handling were discussed while development. Two solutions with different pros and cons are discussed here:

- **Multiple buffer**

Normally each service handler (MainHandler routine) has the whole diagnostic buffer available (apart from the protocol header bytes hidden by CANdesc). Based on this logic the service $22 using PID pipelining has the same tasks as the normal service processor: executing a PID handler and provide him the whole diagnostic buffer for response data. This will hide the whole process and makes the application's life easier (no exceptions for the implementation). To realize this concept means to provide a separate diagnostic buffer for each PID which size is the same as the main one (configured by GENtool). This is a fast and quite simple solution but requires too much RAM to be reserved for only the case

that sometimes the testers would like to use the maximum capacity of the ECU (i.e. requests as many PIDs as possible for this ECU in a single request).

Pros: less ROM usage

Cons: very high RAM usage

- **virtual multiple buffer**

This concept is more generically designed and will not have additional ROM overhead if the pipeline size will be increased. An intelligent buffer concept gives the application the whole size of the buffer for each MainHandler call.

Once the whole data for the current PID has been written, the data supplement will stop (because the next PID handler will not be called). The transmission in the transport layer is started and some time later it runs into buffer under-run. This 'signal' is used to call the next PID MainHandler. This MainHandler has to provide his data very quick. Otherwise the response transmission will stop (due to a continuously buffer under-run).

Pros: less RAM usage (practically independent of the maximum list size).

Cons: moderate ROM overhead / the response data must be composed very quickly.

The virtual multiple buffer concept is the implemented solution. The application can choose for each PID separately to write the data linearly or by using the ring buffer.

performance requirements

The application has performance requirements:

- If linear access has been chosen, the whole response data of each MainHandler must be filled within the lower duration of the P2 time and the TP confirmation timeout. Normally the P2 time is shorter than the transport layers confirmation timeout so just take into account that each Main-Handler must be able to fill its response data within a time far shorter than the P2 time.
- If ring buffer access has been chosen, the application has to call the "DescRingBufferWrite" fast enough to keep TP from confirmation timeout.

Negative response on PID

The negative response handling **is changed** in the multiple PID mode! This affects all protocol-services with a activated 'May be combined' property. The UDS specification encloses only the SIDs: $22 and $2A. For all other services the negative response handling is not changed!

If the application has to reject a request (e.g. ignition key check) it has to do that in the PreHandler. The application is **not allowed** to call "DescSetNegResponse()" to send a negative response in any MainHandler.

This limitation is based on the concept to check all reject conditions in PreHandlers before starting the transmission. This is necessary because after CANdesc has executed the first MainHandler (which starts the positive response transmission) there will be no chance to send a negative response.

The usage of the concept: CANdesc starts to call all PreHandlers of this multiple PID request. If no negative response is set, CANdesc will start to call the corresponding MainHandlers. Within the first call of DescProcessingDone() the transmission is initiated.

Note (for version 3.02.00 of CANdesc and above):

In case the application sets an error code during the main-handler execution in non-debug (released) version of the component, depending on the situation will lead to:

For service $22:

- First DID of the list main-handler: sending a negative response to service $22;
- Second or any of the succeeding DIDs in the list: transmission interruption.

For service $2A:

- Ignoring the scheduled response.

5.3.3.2.1 Sending a positive response

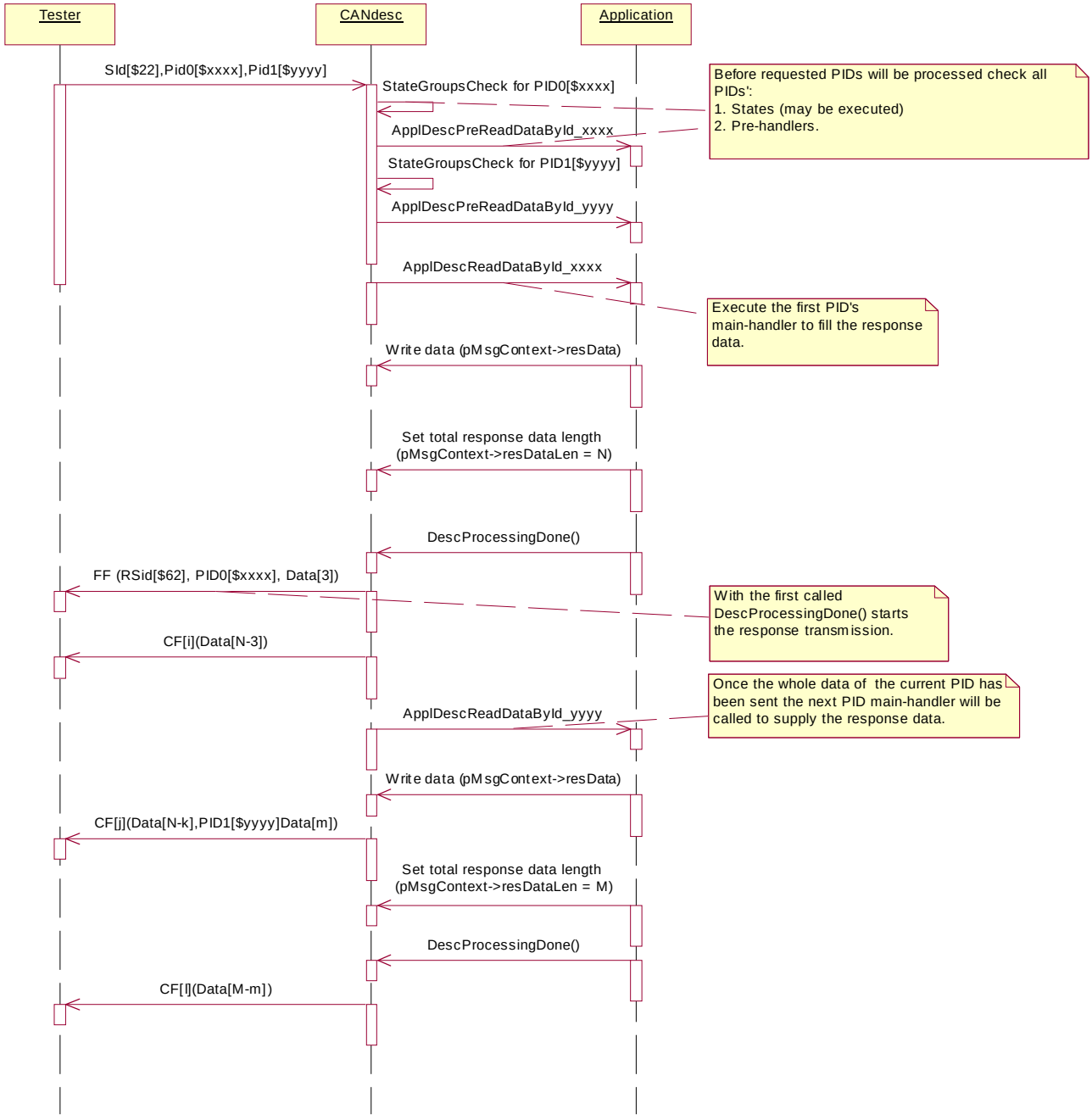


Figure 5-7: Linearly written response data on multiple PIDs (global ring buffer option is on)

5.3.3.2.2 Sending a negative response

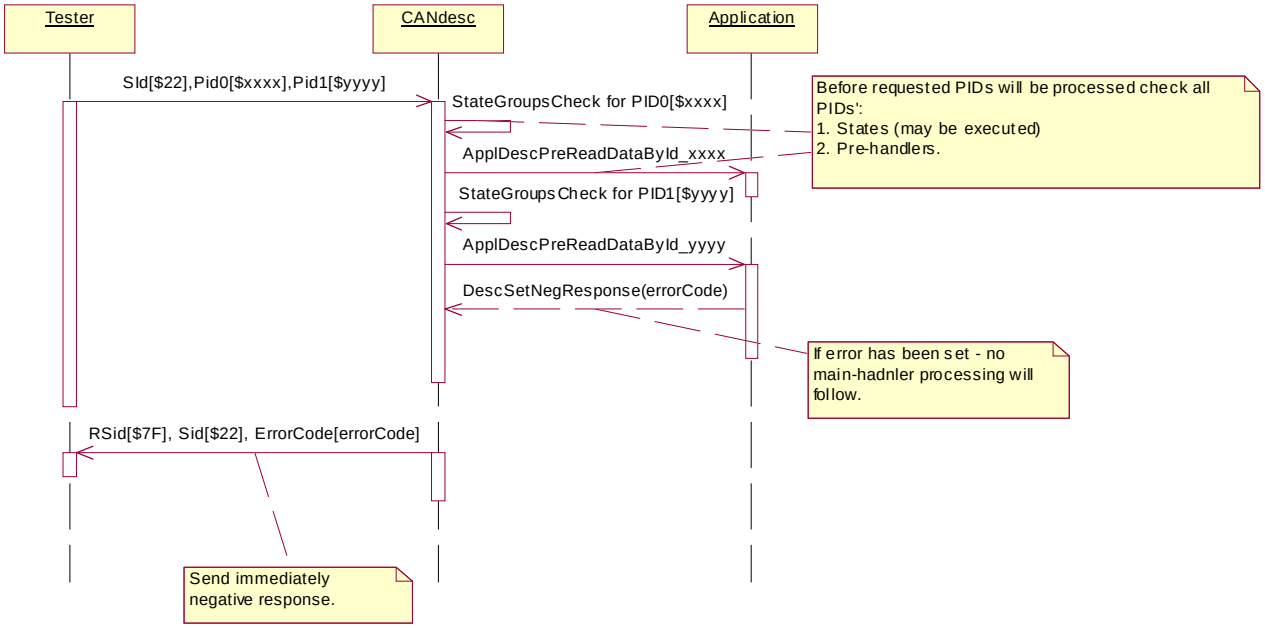


Figure 5-8: Negative response on multiple PIDs (global ring buffer option is on)

5.3.3.2.3 PostHandler execution rule

All PostHandlers are executed after the finished response transmission (like a normal PostHandler).

Independent of the ring-buffer option setting (enabled or disabled), the execution of the service $22 PostHandler(s) has the following rule which has to be taken into account: **calling the Post-Handler of a specific PID means: either the PreHandler of this PID has been previously called or its MainHandler.**

The following sequence chart depicts this:

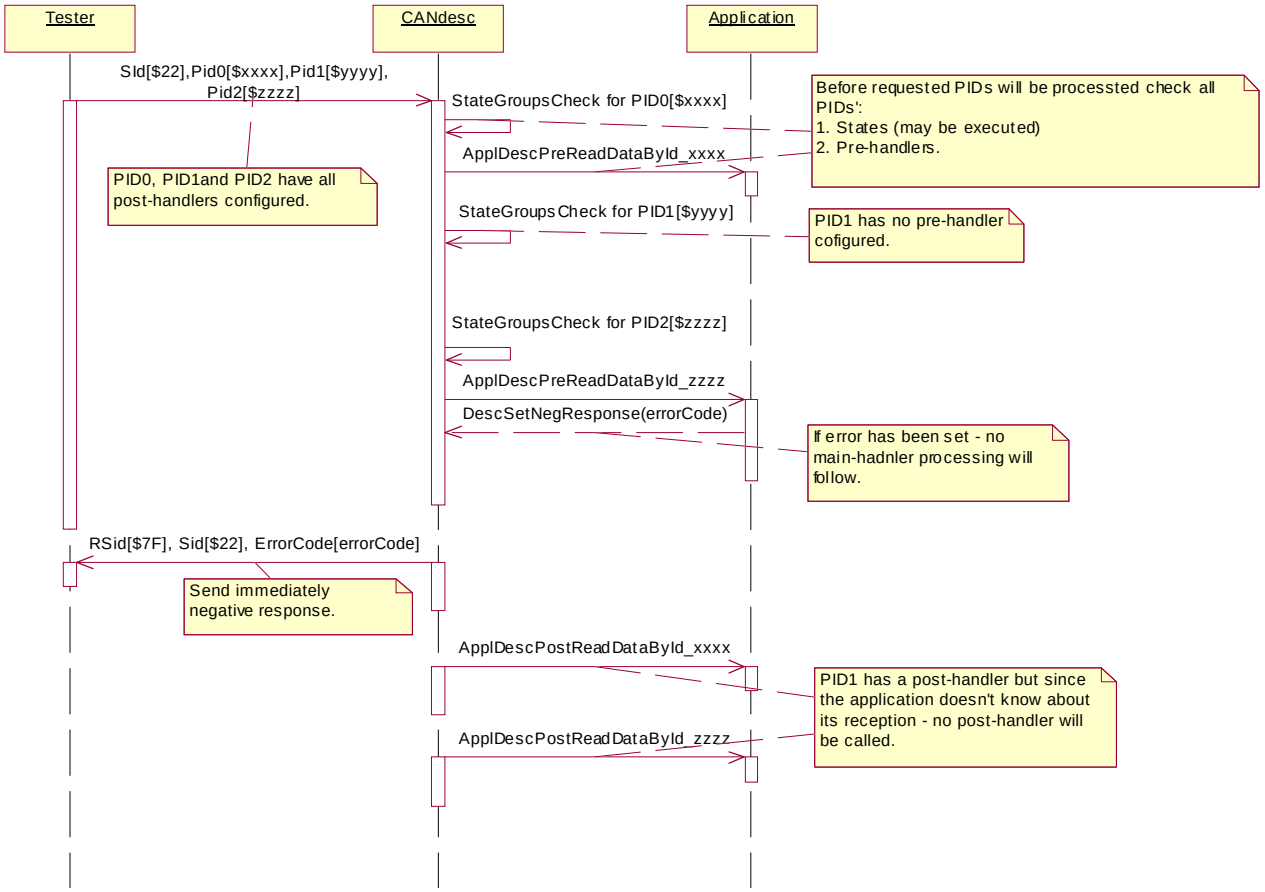


Figure 5-9: Post-Handler execution sequence.

5.4 DynamicallyDefineDataIdentifier (SID $2C) (UDS)

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

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

- a source data identifier, a position and size or,
- a memory address and a memory length, or,
- a combination of the two methods listed above using multiple requests to define the single data element. The dynamically defined dataIdentifier will then contain a concatenation of the data parameter definitions.

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

At last the dynamically defined data identifier consists of a list of (non-dynamically) defined data identifiers and memory area ranges that can be used in any combination.

For more information, see /ISO 14229-1/

5.4.1 Feature set

These are the supported subfunctions for service $2C (DynamicallyDefineDataIdentifier):

Subfunction Name	Hex Value
defineByIdentifier	01
defineByMemoryAddress	02
clearDynamicallyDefinedDataIdentifier	03

5.4.2 API Functions

The reception of a Service $2C request will either delete a DynamicDataIdentifier (DDID) or PeriodicDataIdentifier (PDID) by subfunction $03 or build a DDID/PDID by (several times) using subfunction $01 and/or $02.

For subfunction $02 (defineByMemoryAddress) there is a new application callback function (see chapter 6.6.12 “DynamicallyDefineDataIdentifier ($2C) (UDS) functions”). It allows the application to permit or deny the extension of the DDID/PDID by accessing the defined memory range. The callback function must check, if the requested memory area is readable for the external Tester and if the current security state of the ECU permits the

extension of the DDID/PDID. See chapter 6.6.12.2 for the full set of checks to be executed.

Please note that later, when reading the DDID by using service $22 (ReadDataByIdentifier), further (security) checks for each element of the DDID's list are executed to verify that e.g. the (then active) security state permits the *reading* of the memory area or DID. These checks (of Service $22 and $23) are done in the traditional sequence of Pre-, Main- and PostHandler.

The reception of a Service $22 request starts a new context in CANdesc. Typically the requested data can not be asked from the application by using one single callback function but must be constructed sequentially by collecting data for each part of the DDID's definition list:

- A requested basic source data identifier (DID) is asked of the application by the respective callback (as for Service $22 request), the result data is stripped down to the defined position and size
- A memory address is read by its defined function (typically the same as used for a Service $23 request) and the defined 'size' bytes are collected.

As recommended from /ISO 14229-1/ to prevent data consistency problems a recursive definition of DDIDs is NOT supported.

The Service $22 response data is collected by splitting the service request into these basic tasks, then running the well known internal functions that were defined for them, collect their results and build up the Service $22 response. Therefore, each of the above tasks starts a new context, executes the defined Pre-, Main- and Post-Handler where Application-Callbacks get data, delivers its result and finally ends its context.

The recursive evaluation of DDIDs enforces the usage of MultiContext mode.

We would like to point out that the described operating sequence above is completely run within CANdesc and totally transparent for the application except for the additional API callback function. Using Service $2C or $2A switches CANdesc to MultiContext mode – if your application isn't prepared to support MultiContext mode (by using the defined macros) you'll get compiler errors about inconsistent argument lists.

5.4.3 Sequence Charts

Service $2C – Define a DDID

The following picture exemplifies the sequence of defining a DDID by several call of Service DynamicallyDefineDataIdentifier ($2C).

In our example the first Service $2C request defines the DDID $F300 to return two independent memory areas. For both areas the callback function ApplDescCheckDynDidMemoryArea() is triggered and in this example the application permits both accesses.

The consecutive Service $2C request extends the DDID $F300 by (some fragments of) the existing DID $F010. As the here executed PreHandler does not set a Negative Response Code, CANdesc considers the extension of the DDID valid and enlarges the DDID definition.

A third Service $2C request tries to extend the DDID $F300 once more by another memory area. In our example the call fails, as the specified memory area ($0000) is not valid for this ECU. The service is negative responded and the previous DDID specification is left untouched.

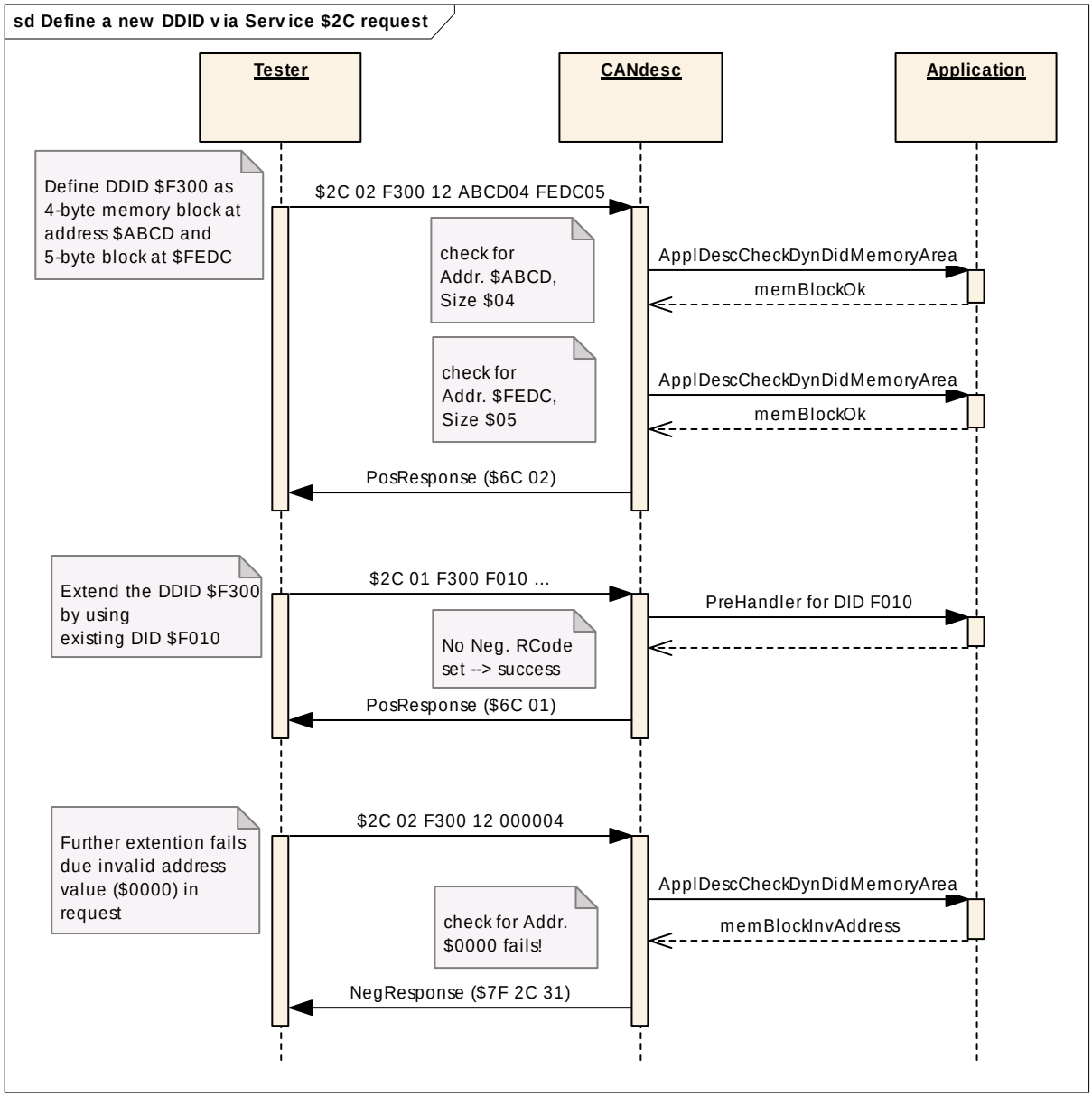


Figure 5-10: Defining a DDID.

Service $22 – Read a DDID

The above defined DDID is now read by Service ReadDataByIdentifier ($22). Within CANdesc the DDID is disassembled into its elements: One (virtual) request for the first memory range, another request for the second memory range and finally a request for the predefined DID $F010.

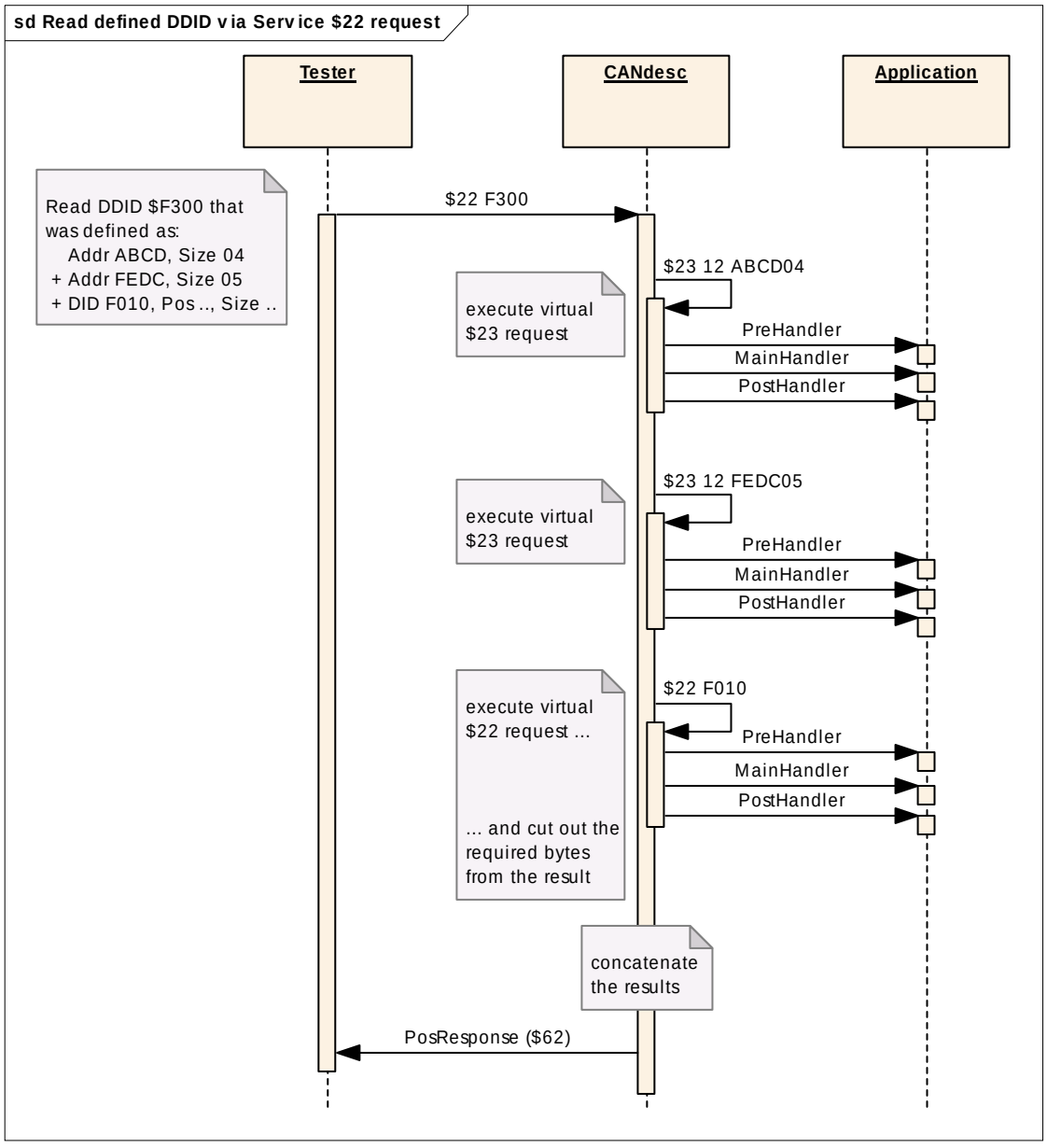


Figure 5-11: Reading a DDID.

Between *CANdesc* and the *application* the sequence looks same as if the tester would have sent 3 requests: (1) ReadMemoryByAddress ($23) on first address range, (2) ReadMemoryByAddress ($23) on second address range, and finally (3) ReadDataByIdentifier ($22) on the DID $F010. Keep in mind: this is just a picture for the succession of events/API-calls - these requests are not real, the messages are never seen on the bus, the internal sequence is actually slightly different but for the application it looks the same!

5.5 Read/Write Memory by Address (SID $23/$3D) (UDS)

Caution
This chapter does not apply to all ECU configurations. Only in special cases the memory access support will be available!

The services $23 (ReadMemoryByAddress) and $3D (WriteMemoryByAddress) are handled uniformly in CANdesc.

Basically the memory by address requests look like this:

$23	FID	*address*	*length*	
$3D	FID	*address*	*length*	*data*

The application need not concern itself with the details how the address and length are formatted. If a valid FID is recognized, CANdesc will extract the address and length information from the request and call an appropriate application callback.

See also:

ApplDescReadMemoryByAddress (6.6.13.1)

ApplDescWriteMemoryByAddress (6.6.13.2)

5.5.1 Tasks performed by CANdesc

To a certain degree CANdesc validates the request.

The basic format checks and service level state validation – this means e.g. security and session validation – are performed before calling the application callback.

Service level state validation means that the request will be denied if all diagnostic instances of service $23 or $3D are not allowed in the current state.

In case of WriteMemoryByAddress the application has linear access to the whole data block to write.

5.5.2 Task to be performed by the Application

CANdesc currently does not provide state validation on format identifier level or memory address / memory block level.

This means, that for example different memory addresses shall require different security levels, the application will have to verify that the ECU currently is in an appropriate state to access the requested memory area.

5.5.3 Repeated service calls

The repeated service call feature is available for the memory access callbacks.

Because they have a different prototype than a normal main handler, the usual API ‘DescStartRepeatedServiceCall (see 6.6.7.1)’ can not be used with the memory access callbacks.

Instead, a new API call ‘DescStartMemByAddrRepeatedCall (see 6.6.7.2)’ has been added.

To abort the repeated service call, use the usual API.

6 CANdesc API

6.1 API Categories

6.1.1 Single Context

This API category is used if no parallel processing is necessary. This is typical for the ISO 14229 specification.

6.1.2 Multiple Context (only CANdesc)

This API category is used if parallel processing is necessary. This means not that CANdesc can work with multiple instances, but only one functional request can be processed parallel to a working physical request.

6.2 Data Types

The following standard data types are used in this document:

vuint8	Represents 8 bit unsigned integer value.
vsint8	Represents 8 bit signed integer value.
vuint16	Represents 16 bit unsigned integer value.
vsint16	Represents 16 bit signed integer value.
vuint32	Represents 32 bit unsigned integer value.
vsint32	Represents 32 bit signed integer value.

Table 6-1: standard data types

Additional data types used in this document are described in the corresponding function description.

6.3 Global Variables

-

6.4 Constants

6.4.1 Component Version

The version of the CANdesc component consist of 3 parts in the following format: **MM.SS.BB**,

Where:

- **MM** is the main version of the component,
- **SS** is the subversion of the component,
- **BB** is the bug-fix version of the component.

To get the current CANdesc version, the application could use the following shared data:

Name	Type	Description
g_descMainVersion	BCD	Contains the main version part.
g_descSubVersion	BCD	Contains the subversion part.
g_descBugFixVersion	BCD	Contains the bug-fix version part.

Table 6-2: Version API data

Note: The version of the module is the same as the version of the generator's DLL file.

6.5 Macros

6.5.1 Data exchange

The CANdesc provides a generic API for splitting a multi-byte (up to 4 bytes) variable to a byte sequence with platform transparent access to each byte, and assembling a multi-byte (up to 4 bytes) variable from a sequence of bytes.

6.5.1.1 Splitting 16 bit data

The following function could be used to get platform independent access to the corresponding bytes of 16 bit data variable:

vuint8 DescGetHiByte(16BitData)

vuint8 DescGetLoByte(16BitData)

6.5.1.2 Splitting 32 bit data

The following function could be used to get platform independent access to the corresponding bytes of 32 bit data variable:

vuint8 DescGetHiHiByte(32BitData)

vuint8 DescGetHiLoByte(32BitData)

vuint8 DescGetLoHiByte(32BitData)

vuint8 DescGetLoLoByte(32BitData)

6.5.1.3 Assembling 16 bit data

The application can create the 16 bit signal from a byte stream using the following API:

uint16 DescMake16Bit(hiByte, loByte)

where the **hiByte, loByte** are the corresponding bytes for the returned 16 bit data.

6.5.1.4 Assembling 32 bit data

The application can create the 32 bit signal from a byte stream using the following API:

uint32 DescMake32Bit(HiHiByte, HiLoByte, LoHiByte, LoLoByte)

where the **HiHiByte, HiLoByte, LoHiByte, LoLoByte** are the corresponding bytes for the returned 32 bit dat

6.6 Functions

6.6.1 Administrative Functions

6.6.1.1 DescInitPowerOn()

DescInitPowerOn
Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void DescInitPowerOn (DescInitParam initParameter)`	
Multi Context	
`void DescInitPowerOn (DescInitParam initParameter)`	
Parameter	
`initParameter`	Manufacturer specific type, please refer ‘CANdesc: OEM specifics’ document
Return code	
-	-
Functional Description	
PowerOn Initialization of the CANdesc. This function has to be called once before all other functions of CANdesc after PowerOn.	
Pre-conditions	
Correctly initialized CAN-driver via **CanInitPowerOn()** and TransportLayer via **TpInitPowerOn()**.	
Call context	
Background-loop level with global disabled interrupts	
Particularities and Limitations	
■ **DescInitPowerOn (initParameter)** must be called after **TpInitPowerOn()** was called (please, refer the /TPMC/ documentation), otherwise the reserved diagnostic connection will be los	

6.6.1.2 DescInit()

DescInit

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void DescInit (DescInitParam initParameter)`	
Multi Context	
`void DescInit (DescInitParam initParameter)`	
Parameter	
`initParameter`	Manufacturer specific type, please refer ‘CANdesc Part IV: OEM specifics’ document
Return code	
-	-
Functional Description	
Re-initialization of CANdesc. This function can be called to re-initialize CANdesc (e.g. after WakeUp). All internal states will be set to default, except the states in this initParameter (e.g. Session or CommunicationControl).	
Pre-conditions	
CANdesc was once initialized via **DescInitPowerOn ()**	
Call context	
Background-loop level with global disabled	

6.6.1.3 DescTask()

DescTask

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void DescTask (void)`	
Multi Context	
`void DescTask (void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
The function ***DescTask()*** has to be called periodically (cycle time $T_{DescCallCycle}$) by the application. Within the context of this function the interaction with the application is performed. In addition the monitoring of the timings is done, therefore the accuracy of the timings depends on the call cycle and on the accuracy of the calls.	
Pre-conditions	
-	
Call context	
Background-loop level or OSEK-OS Task. The task should have a lower or equal priority than all other interaction to the CANdesc component.	
Particularities and Limitations	
■ May not be called if the ***DescStateTask()*** and ***DescTimerTask()*** are called. ■	

6.6.1.4 DescStateTask()

DescStateTask

Available since 4.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void DescStateTask (void)`	
Multi Context	
`void DescStateTask (void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	

Motivation: Using a single task function for timers and processing leads either to slow processing or to faster timers which costs runtime for the ECU. The timers need very stable cyclical call but the processing tasks may be done “as soon as possible” (i.e. using OSEK to be assigned to lower priority task).

The function ***DescStateTask()*** has to be called periodically by the application. It is not a timer task – it has no specific time period. As smaller this tasks call period is, so faster will be the service processing.

This task function will process received request and to control the transmission of the responses. Depending on the ECU requirements it is recommended to call this task as soon as possible to avoid delays of the response (e.g. dynamically defined DID, scheduled data, etc.), but **take into account that within this task the corresponding MainHandler will be executed too.**

Pre-conditions
-
Call context
Background-loop level or OSEK-OS Task. The Task should have a lower or equal priority than all other interaction to the CANdesc component.
Particularities and Limitations

- May not be called if the ***DescTask()*** is used (reentrancy is forbidden).
-

6.6.1.5 DescTimerTask()

DescTimerTask

Available since 4.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void DescTimerTask (void)`	
Multi Context	
`void DescTimerTask (void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
Motivation: Using a single task function for timers and processing leads either to slow processing or to faster timers which costs runtime for the ECU. The timers need very stable cyclical call but the processing tasks may be done “as soon as possible” (i.e. using OSEK to be assigned to lower priority task). The function ***DescTimerTask()*** has to be called periodically by the application in the configured task period. It can be called as slow as possible to free run time resources.	
Pre-conditions	
-	
Call context	
Background-loop level or OSEK-OS Task. The Task should have a lower or equal priority than all other interaction to the CANdesc component.	
Particularities and Limitations	
■ May not be called if the ***DescTask()*** is used. This will lead to either reentrancy (consistency) problems or/and to timing issues. ■	

6.6.1.6 DescGetActivityState()

DescGetActivityState
Available since 2.00.00
Is Reentrant ☒
Is callback ☐

Prototype	
Single Context	
`DescContextActivity DescGetActivityState (void)`	
Multi Context	
`DescContextActivity DescGetActivityState (vuint8 iContext)`	
Parameter	
`iContext`	reference to the corresponding request context
Return code	
1. `kDescContextIdle` 2. `kDescContextActiveRxBegin` 3. `kDescContextActiveRxEnd` 4. `kDescContextActiveProcess` 5. `kDescContextActiveProcessEnd` 6. `kDescContextActiveTxReady` 7. `kDescContextActiveTx` 8. `kDescContextActivePostProcess`	1. There is currently no request processing (even when scheduler is active). 2. Currently request reception is active. 3. Reception finished, request will be processed. 4. The request was received, is under processing now 5. DescProcessingDone called waiting for data before starting the transmission. 6. Ready for response transmission. 7. Transmission of the response is currently active. 8. Transmission/processing ended. Post-processing will be performed.
Functional Description	
Motivation: Sometimes the knowledge about the presence of a tester is necessary. A typical use-case is to avoid the ECU from going into sleep mode. A non-default session indicates that a tester is present. But how can this be done, if the ECU is in the default session? Due to that fact the ECU application can call the function ***DescGetActivityState()*** any time to check if CANdesc has something to do or is in idle mode. This can be used e.g. to change the state of the ECU sleep mode. **Note:** The return value is bit coded and any senseful combination of the above mentioned values is possible (e.g. ***kDescContextActiveRxBegin*** | ***kDescContextActivePostProcess***). Please check always with bit test (and operation) and not using the value comparison.	
Pre-conditions	
-	
Call context	
-	
Particularities and Limitations	

6.6.2 Service Functions

6.6.2.1 DescSetNegResponse()

DescSetNegResponse
Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void DescSetNegResponse (DescNegResCode errorCode)`	
Multi Context	
`void DescSetNegResponse (vuint8 iContext, DescNegResCode errorCode)`	
Parameter	
`iContext`	reference to the corresponding request context
`errorCode`	the **errorCode** is the one of the provided error code constants of CANdesc in the desc.h file with the following naming convention: **kDescNrc<error name>**.
Return code	
-	-
Functional Description	
In the PreHandler or in the MainHandler function the application has the possibility of forcing negative response with a certain negative response code for the current request when it is necessary.	
Pre-conditions	
-	
Call context	
Within a ‘Service PreHandler’ function and within or after a ‘Service MainHandler’ function	
Particularities and Limitations	
■ Once an error was set it can not be overwritten or reset. ■ This function does not finish the processing of the request. It just sets a certain error and after that the application must confirm that the request processing was completely finished by calling DescProcessingDone().	

6.6.2.2 DescProcessingDone()

DescProcessingDone
Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void` **`DescProcessingDone`** `(void)`	
Multi Context	
`void` **`DescProcessingDone`** `(vuint8 iContext)`	
Parameter	
`iContext`	reference to the corresponding request context
Return code	
-	-
Functional Description	
After completing the request execution the application must call the API function. By calling this function, depending on the previous actions of the application the CANdesc module will either send a response (positive/negative depending on the error state machine) or no response will be send if the application/CANdesc decides that there must be no response (please refer the Part III User Manual)	
Pre-conditions	
-	
Call context	
Within or after a ‘Service MainHandler’ function	
Particularities and Limitations	

6.6.3 Service Call-Back functions

6.6.3.1 Service PreHandler

ApplDescPre<Service-Qualifier + Instance-Qualifier>>

Available since 2.00.00
Is callback ☒

Prototype	
Single Context	
`void` **`ApplDescPre<Service-Qualifier + Instance-Qualifier>`** `(void)`	
Multi Context	
`void` **`ApplDescPre<Service-Qualifier + Instance-Qualifier>`** `(vuint8 iContext)`	
Parameter	
`iContext`	the current request context location
Return code	
-	-
Functional Description	
The PreHandler is executed before the Service MainHandler is called. In the PreHandler, the application can hook any (especially application-specific) state validations. One PreHandler implementation may be shared with different service instances (only CANdesc). To allow quite complex operations to take place, the application has access to the request data using the context data structure (if given).	
Pre-conditions	
Must be configured to ‘User’ in attribute ‘PreHandlerSupport’’	
Call context	
From **DescTask()**	
Particularities and	

6.6.3.2 Service MainHandler

ApplDesc<Service-Qualifier + Instance-Qualifier>

Available since 2.00.00
Is callback ☒

Prototype	
Single Context	
`void` **`ApplDesc<Service-Qualifier + Instance-Qualifier>`** `(DescMsgContext* pMsgContext)`	
Multi Context	
`void` **`ApplDesc<Service-Qualifier + Instance-Qualifier>`** `(DescMsgContext* pMsgContext)`	
Parameter	

`pMsgContext`	`typedef struct`
	`{`
	`  DescMsg  reqData;`
	`  DescMsgLen  reqDataLen;`
	`  DescMsg  resData;`
	`  DescMsgLen  resDataLen;`
	`  DescMsgAddInfo  msgAddInfo;`
	`  vuint8  iContext;`
	`  t_descUsdtNetBus busInfo;`
	`} DescMsgContext;`

`DescMsgAddInfo`	`  DescBitType reqType  :2; /* 0x01: Phys 0x02: Func */`
	`  DescBitType resOnReq  :2; /* 0x01: Phys 0x02: Func */`
	`  DescBitType suppPosRes:1; /* 0x00: No  0x01: Yes */`

`Read access`	**pMsgContext->reqData** pointer to the first byte of the already extracted request data. **pMsgContext->reqDataLen** length of the extracted request data. **pMsgContext->iContext** the current request context location (used only as a handle - *DO NOT MODIFY*). **pMsgContext->msgAddInfo.reqType** the current request addressing method. Could be either ‚kDescFuncReq' or ‚kDescPhysReq' (bitmapped). **pMsgContext->msgAddInfo.suppPosRes** if set, no positive response will be sent. (UDS only). **pMsgContext->busInfo** the current request communication information (i.e. driver type (CAN, MOST, FlexRay, etc.), addressing information, communication channel number, tester address (if applicable) etc.
`Write access`	**pMsgContext->resData** pointer to the first position where the response data can be written. **pMsgContext->resDataLen** length of the written data. **pMsgContext->msgAddInfo.resOnReq** can be used to disable the response transmission on the current request. If set to '0' no response will be transmitted. Physical and function can be set separately (bitmapped).
Return code	

-	-

Functional Description

The MainHandler processes the service request.

- Perform length validation for varying length information of request.
- Disassemble any data received with the request telegram and process it,.
- Assemble any data to be send with the response and update current response length.
- Confirm that the processing is finished.

Pre-conditions

Must be configured to ‘User’ in attribute ‘MainHandlerSupport’

Call context

From **DescTask()**

Particularities and Limitations

- If used as MainHandler for Protocol Services, the Protocol-Service-Qualifier is used instead
-

6.6.3.3 Service PostHandler

ApplDescPost<Service-Qualifier + Instance-Qualifier>

Available since 2.00.00
Is callback ☒

Prototype	
Single Context	
`void` **`ApplDescPost<Service-Qualifier + Instance-Qualifier>`** `(vuint8 status)`	
Multi Context	
`void` **`ApplDescPost<Service-Qualifier + Instance-Qualifier>`** `(vuint8 iContext, vuint8 status)`	
Parameter	
`iContext`	the current request context location
`status (bit-coded)`	**KDescPostHandlerStateOk** The positive response was transmitted successfully **KDescPostHandlerStateNegResSent** It was a negative response **kDescPostHandlerStateTxFailed** A transmission error occurred
Return code	
-	-

Functional Description

Any state transition may not be performed before the current service is finished completely (the last frame of the response is sent successfully).

The PostHandler is executed after a confirmation of the message transmission is received and is designated for state adaptation – all other things are already done when the PostHandler is called.

Pre-conditions

Must be configured to ‘User’ in attribute ‘PostHandlerSupport’

Call context

From **DescTask()**

Particularities and Limitations

- If used as PostHandler for Protocol Services, the Protocol-Service-Qualifier is used instead
- You can override the given name extension (Service-Qualifier + Instance-Qualifier) by using the ‘PostHandlerOverrideName’.

6.6.4 User (Unknown) Service Handling

In some cases the ECU shall support a service which is not described in the common way for CANdesc (by means of CANdelaStudio/GENtool). With a little bit more effort inside the application than for the “known” services the ECU is still be able to support those user defined services. The effort comes form the fact that CANdesc knows nothing about this service (e.g. session, security or other states described in the CDD configuring CANdesc, addressing methods allowed for those services, etc.) and therefore the application must do this work for each user defined service by itself. In fact for CANdesc there is only one “unknown” service and it is up to the application to differentiate between multiple unknown service(s).

Attention: This feature is available since version 2.11.00 of CANdesc(Basic).

6.6.4.1 How it works

If the feature “Support Generic User Service” is enabled in the GENtool CANdesc uses following handling:

- if a service was not recognized by its SID, before the automatic negative response transmission will be sent, the application will be called (see 6.6.4.2 ApplDescCheckUserService) to check this SID too. If it can not recognize it as a valid one the usual negative response will be sent.
- If the application has accepted the SID, then a special “user service” MainHandler will be called (see 6.6.4.4 Generic User Service MainHandler).
- If in GENtool “Support Generic User Service PostHandler” is set, after the request processing has been accomplished, a special “user service” PostHandler will be called (see 6.6.4.5 Generic User Service PostHandler).

Note:

- Since CANdesc doesn’t distinguish user defined services, a special API was designed to get the application the opportunity to dispatch among the SIDs (in MainHandler and in the PostHandler).
- The user defined services are processed on service id level which means the application shall dispatch and do the whole format check of these requests. The state management shall be performed bye application, too.

6.6.4.2 ApplDescCheckUserService()

ApplDescCheckUserService

Available since 2.11.00
Is callback ☒

Prototype	
Single Context	
`vuint8 ApplDescCheckUserService (DescMsgItem sid)`	
Multi Context	
`vuint8 ApplDescCheckUserService (DescMsgItem sid)`	
Parameter	
`sid`	The service identifier which is currently under processing.
Return code	
1. `kDescOk` 2. `kDescFailed`	1. Return this value if the service id is a “user defined” one. 2. Return this value if the service id is unknown for the application too.
Functional Description	
The currently received request contains an unknown for CANdesc service Id. Within this function the ECU application has to decide immediately if the SID is one of the user defined or not. Depending on the return value, CANdesc will process further this request or will reject it by sending negative response ‘ServiceNotSupported’.	
Pre-conditions	
The “Support Generic User Service” option was enabled in the GENtool configuration.	
Call context	
From **DescTask()** (in KWP diagnostics also from RxInterrupt).	
Particularities and Limitations	

6.6.4.3 DescGetServiceId()

DescGetServiceId

Available since 2.11.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`DescMsgItem` **`DescGetServiceId`** `(void)`	
Multi Context	
`DescMsgItem` **`DescGetServiceId`** `(vuint8 iContext)`	
Parameter	
`iContext`	The current request context location
Return code	
`DescMsgItem`	The service id which is currently under processing.
Functional Description	
Reports the service id of the currently processed user-service request.	
Pre-conditions	
The “Support Generic User Service” option was enabled in the GENtool configuration.	
Call context	
From **DescTask()**	
Particularities and Limitations	
■ This function may be called at any time within a diagnostic request life cycle starting at the call of the MainHandler and ending by the PostHandler (if configured) or (if none configured) by calling **DescProcessingDon**	

6.6.4.4 Generic User Service MainHandler

ApplDescUserServiceHandler

Available since 2.11.00
Is callback ☒

Prototype	
Single Context	
`void ApplDescUserServiceHandler (DescMsgContext* pMsgContext)`	
Multi Context	
`void ApplDescUserServiceHandler (DescMsgContext* pMsgContext)`	
Parameter	
`pMsgContext`	Refer the section 6.6.3.2 Service MainHandler for details about this parameter.
`Read Access`	**pMsgContext->reqData** pointer to the first byte after the service Id. The other members of the parameter are described in 6.6.3.2 Service MainHandler
`Write access`	**pMsgContext->resData** pointer to the first byte after the response SID, where the data (incl. sub-parameters) will be written. The other members of the parameter are described in 6.6.3.2 Service MainHandler
Return code	
-	-

Functional Description

This MainHandler is called for all unknown service requests at service id level, so the application has to do following:

- Perform service id dispatching (if more than one user defined service shall be used).
- Perform length validation for varying length information of request.
- Perform parameter (if any) validation.
- Disassemble any data received with the request telegram and process it.
- Assemble any data to be send with the response and update current response length
- Confirm that the processing is finished.

Pre-conditions

The “Support Generic User Service” option was enabled in the GENtool configuration.

Call context

From **DescTask()**

Particularities and Limitations

- Refer the section 6.6.3.2 Service MainHandler.
- **DescGetServiceId()** may be called here to dispatch the SID of the currently processed user service (refer 6.6.4.3 DescGetServiceId

6.6.4.5 Generic User Service PostHandler

ApplDescPostUserServiceHandler

Available since 2.11.00
Is callback ☒

Prototype	
Single Context	
`void ApplDescPostUserServiceHandler (vuint8 status)`	
Multi Context	
`void ApplDescPostUserServiceHandler (vuint8 iContext, vuint8 status)`	
Parameter	
`iContext, status`	Refer 6.6.3.3 Service PostHandler for information.
Return code	
-	-
Functional Description	
The functionality of the user service PostHandler is the same as the one of the normal service PostHandler. Refer 6.6.3.3 Service PostHandler for more details.	
Pre-conditions	
The “Support Generic User Service PostHandler” option was enabled in the GENtool configuration. CANdesc version >= 2.11.00	
Call context	
From **DescTask()**	
Particularities and Limitations	
■ Refer the section 6.6.3.3 Service PostHandler for information. ■ **DescGetServiceId()** may be called here to dispatch the SID of the currently post-processed user service (refer 6.6.4.3 DescGetServiceId	

6.6.5 Session Handling

6.6.5.1 ApplDescCheckSessionTransition()

ApplDescCheckSessionTransition

Available since 2.00.00
Is callback ☒

Prototype	
Single Context	
`void` **`ApplDescCheckSessionTransition`** `(DescStateGroup newState, DescStateGroup formerState)`	
Multi Context	
`void` **`ApplDescCheckSessionTransition`** `(vuint8 iContext, DescStateGroup newState, DescStateGroup formerState)`	
Parameter	
`iContext`	the current request context location
`newState`	the CANdesc component has change to this session state
`formerState`	the CANdesc component has change from this session state
Return code	
-	-

Functional Description

This hook function will be called, while session request is received (SID $10). If the application wants to discard this request, an error must be set (via **DescSetNegResponse()**).

The application always has to confirm this hook function via **DescSessionTransitionChecked().**

Both above functions can be called also outside of the context of this function (e.g. application task waiting for results form an I/O port). CANdesc will send RCR-RP response as long as the application delays the confirmation for the session transition.

In some cases the application has to know whether the SPRMIB in the request was set or not. Since this API call does not contain this information, a dedicated API in CANdesc provides it: *DescIsSuppressPosResBitSet ()*.

Pre-conditions

At least one DiagnosticSessionControl service must be configured to ‘OEM’ in attribute ‘MainHandlerSupport’

Call context

From **DescTask()**

Particularities and Limitations

- Call the API function **DescSessionTransitionChecked()** to end the service processing
-

6.6.5.2 DescSessionTransitionChecked()

DescSessionTransitionChecked

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void` **`DescSessionTransitionChecked`** `(void)`	
Multi Context	
`void` **`DescSessionTransitionChecked`** `(vuint8 iContext)`	
Parameter	
`iContext`	the current request context location
Return code	
-	-
Functional Description	
After the application has finished the processing in the hook function **ApplDescCheckSessionTransition()** this function must be called.	
Pre-conditions	
At least one DiagnosticSessionControl service must be configured to ‘OEM’ in attribute ‘MainHandlerSupport’	
Call context	
Within or after a ‘**ApplDescCheckSessionTransition()**’ function	
Particularities and Limitations	
■ If this function will be called late, the CANdesc component sends automatically the RCR-RP responses	

6.6.5.3 DescIsSuppressPosResBitSet ()

DescIsSuppressPosResBitSet

Available since 5.07.14
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`DescBool` **`DescIsSuppressPosResBitSet`** `(void)`	
Multi Context	
`DescBool` **`DescIsSuppressPosResBitSet`** `(vuint8 iContext)`	
Parameter	
`iContext`	the current request context location
Return code	
`kDescTrue`	The SPRMIB is set.
`kDescFalse`	The SPRMIB is NOT set.
Functional Description	
This API can be always called while a diagnostic service processing is ongoing to get the information about the SPRMIB state. All main-handlers do contain this information already in the pMsgContext parameter so use it instead of this API. In some other cases the application does not have access to the pMsgContext, and there the API can be used.	
Pre-conditions	
Only for UDS configurations. May be called only while a diagnostic service processing is ongoing. Otherwise invalid data can be reported.	
Call context	
Any.	
Particularities and Limitations	
■	

6.6.5.4 ApplDescOnTransitionSession()

ApplDescOnTransitionSession

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void ApplDescOnTransitionSession (DescStateGroup newState, DescStateGroup formerState)`	
Multi Context	
`void ApplDescOnTransitionSession (DescStateGroup newState, DescStateGroup formerState)`	
Parameter	
`newState`	the CANdesc component has change to this session state
`formerState`	the CANdesc component has change from this session state
Return code	
-	-
Functional Description	
After the positive response of a SessionControl request the session will transit to the requested session. This function informs the application that such a transition occurs.	
Pre-conditions	
-	
Call context	
From **DescTask()** interrupts might be disabled	
Particularities and Limitations	
■ Only informational function	

6.6.5.5 DescSetStateSession()

DescSetStateSession

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void` **`DescSetStateSession`** `(DescStateGroup newSession)`	
Multi Context	
`void` **`DescSetStateSession`** `(DescStateGroup newSession)`	
Parameter	
`newSession`	the CANdesc component will change to this session state
Return code	
-	-
Functional Description	
By this function the state of the SessionState-group can be changed by the ECU application. The transition notification function 'ApplDescOnTransitionSession' will be called to notify the application about the new session.	
Pre-conditions	
-	
Call context	
-	
Particularities and Limitations	
■ Refer the section 6.6.10.2 "DescSetState<StateGroup>()" for more details.	

6.6.5.6 DescGetStateSession()

DescGetStateSession
Available since 2.00.00
Is Reentrant ☒
Is callback ☐

Prototype	
Single Context	
`currentSession` **`DescGetStateSession`** `(void)`	
Multi Context	
`currentSession` **`DescGetStateSession`** `(void)`	
Parameter	
-	
Return code	
`currentSession`	
Functional Description	

This function returns the current session state. Since the states are bit-coded the evaluation expressions may be optimized for multiple use cases.

Example: Code execution only when either default or extended session is active.

```
lState = DescGetStateSession();
if ( (lState & (kDescStateSession<Default>) | kDescStateSession<Extended>)) != 0 )
{
 /*execute code*/
}
```

Pre-conditions
-
Call context
-
Particularities and Limitations

- Refer the section 6.6.10.1 “DescGetState<StateGroup>()” for more details.

6.6.6 CommunicationControl Handling

This API is provided, if the ECU supports the serviceCommunicationControl (UDS) or service 0x28/0x29 Dis-/EnableNormalMessageTransmission (KWP).

6.6.6.1 ApplDescCheckCommCtrl()

ApplDescCheckCommCtrl

Available since 2.00.00
Is callback ☒

Prototype	
Single Context	
`void ApplDescCheckCommCtrl (DescOemCommControlInfo* commControlInfo)`	
Multi Context	
`void ApplDescCheckCommCtrl (vuint8 iContext, DescOemCommControlInfo* commControlInfo)`	
Parameter	
`iContext`	The current request context location
`commControlInfo`	OEM dependent
Return code	
-	-
Functional Description	
The execution of this service is completely done within the CANdesc component. This hook function can be used to permit the application to reject the execution under some circumstance. If the application wants to discard this request, an error must be set (via **DescSetNegResponse()**). The application always has to confirm this hook function (via **DescCommCtrlChecked()**).	
Pre-conditions	
The CommunicationControl service must be activated and the attribute 'MainHandlerSupport' has to be set to 'OEM'	
Call context	
From **DescTask()**	
Particularities and Limitations	
■ If the API function **DescCommCtrlChecked()** will be not called, the service processing will not end	

6.6.6.2 DescCommCtrlChecked()

DescCommCtrlChecked

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void DescCommCtrlChecked (void)`	
Multi Context	
`void DescCommCtrlChecked (vuint8 iContext)`	
Parameter	
`iContext`	the current request context location
Return code	
-	-
Functional Description	
The CANdesc component calls a hook function to check for the execution permission of the CommunicationControl service. Within or after this hook function (**ApplDescCheckCommCtrl()**) the application can set an error (**DescSetNegResponse()**) to reject the request. This function is used to terminate the hook function **ApplDescCheckCommCtrl().**	
Pre-conditions	
The CommunicationControl service must be activated and the attribute ‘MainHandlerSupport’ has to be set to ‘OEM’	
Call context	
Within or after **ApplDescCheckCommCtrl()**	
Particularities and Limitations	
■	

6.6.7 Periodic call of ‘Service MainHandler’

6.6.7.1 DescStartRepeatedServiceCall()

DescStartRepeatedServiceCall

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void DescStartRepeatedServiceCall (DescMainHandler descMainHandler)`	
Multi Context	
`void DescStartRepeatedServiceCall (vuint8 iContext, DescMainHandler descMainHandler)`	
Parameter	
`descMainHandler`	Reference to a function. The function prototype must be based on a ‘Service MainHandler’.
`iContext`	The current request context location
Return code	
-	-
Functional Description	
The application can use this function to get a periodic call to the specified function (in the parameter) from the CANdesc component. It is possible to use the same ‘Service MainHandler’ function as it is called in.	
Pre-conditions	
Call context	
Within or after a ‘Service MainHandler’ function	
Particularities and Limitations	
■ CANdesc can do no validation, if this pointer is valid. ■ Is the parameter NULL, the periodic calls will get stopped. ■ The function is called in the same cycle time (context) as the DescTask()	

6.6.7.2 DescStartMemByAddrRepeatedCall()

DescStartMemByAddrRepeatedCall

Available since 5.06.04
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void` **`DescStartMemByAddrRepeatedCall`** `()`	
Multi Context	
`void` **`DescStartMemByAddrRepeatedCall`** `(vuint8 iContext)`	
Parameter	
`iContext`	The current request context location
Return code	
-	-
Functional Description	
The application can use this function to get a periodic call to the current Read/Write memory by address handler.	
Pre-conditions	
Call context	
Within ApplDescReadMemoryByAddress or ApplDescWriteMemoryByAddress.	
Particularities and Limitations	
■ The memory access handler is called in the same cycle time (context) as the DescTask()	

6.6.8 Ring Buffer Mechanism

The ring-buffer option can be used to save RAM when some responses are quite long and reserving such space of RAM is impossible. In contrast to the linear responses, where the response data will be first written and then the transmission to the tester will be initiated, the ring-buffer concept starts a transmission as soon as it has either the whole data (for short [single frame] responses) or at least enough data to fill a first-frame of a multi-frame transmission. Once the ring buffer has been activated and the response transmission initiated the application must supply enough data to keep the transmission away from lack of data. Therefore the ring-buffer can not be used in diagnostic services which allow multiple data to be combined in a single request (e.g. in CANdelaStudio the flag "*multiple identifiers of different instances may be combined in one request*" is set). Such services are existing in both **KWP 14230 (OBD)** and the **UDS 14229OBD, ReadDataByIdentifier ($22), ReadDataByPeriodicIdentifier ($2A))** standard.

Caution

On UDS: Always check the SPRMIB prior starting the ring-buffer. If this bit is set, the ring-buffer may not be started. Instead the API **DescProcessingDone()** must be called. The response length can be set to zero since there will be no response on the bus.

6.6.8.1 DescRingBufferStart()

DescRingBufferStart

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void DescRingBufferStart (void)`	
Multi Context	
`void DescRingBufferStart (vuint8 iContext)`	
Parameter	
`iContext`	reference to the corresponding request context
Return code	
-	-

Functional Description

After completing the request validation the application can decide (in runtime), if the ring-buffer mechanism should be used or not.

By calling this function, the decision is made to use the ring-buffer. Otherwise DescProcessingDone() should be called, after filling the response data (in a linear way). Either DescProcessingDone() or DescRingBufferStart() will finish the response handling.

Depending on the previous actions of the application the CANdesc module will either send a response (positive/negative depending on the error state machine) or no response will be send if the application/CANdesc decides that there must be no response (please refer the Part III User Manual).

The transmission of the positive response will not start immediately. The application has to fill the ring-buffer first. If the ring-buffer has enough data, the transmission will be started (internally).

Pre-conditions

- ring-buffer has been enabled in the configuration

Call context

Within or after a ‘Service MainHandler’ function

Particularities and Limitations

- This API **must not** be called from any of the other handler type (Pre- or PostHandlers)
- Either DescProcessingDone() or DescRingBufferStart() must be used to finish the response handling.
- Total response length must be written before!
- No response data must be written before!
- This function **must not** be called in interrupt context
- **Limitation:** Until CANdesc version 2.13.00 it was not possible to use the Ring-Buffer in ‘Multiple PID’ services (as described in section 5.3.3 Multiple PID mode)
- **UDS limitation:** Always check the SPRMIB prior starting the ring-buffer. If this bit is set, the ring-buffer shall not be started. Instead DescProcessingDone() must be called (see 7.6).

6.6.8.2 DescRingBufferWrite()

DescRingBufferWrite

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`vuint8 DescRingBufferWrite (DescMsg data, DescMsgLen dataLength)`	
Multi Context	
`vuint8 DescRingBufferWrite (vuint8 iContext, DescMsg data, DescMsgLen dataLength)`	
Parameter	
`iContext`	Reference to the corresponding request context
`DescMsg`	Pointer to application data, which should be copied into ring-buffer.
`DescMsgLen`	Amount of data, which should be copied (from pointer data) into ring-buffer.
Return code	
`vuint8`	**kDescOk** If the copy process was successful **kDescFailed** if the data are **not** copied into the ring-buffer
Functional Description	
The application writes data into the ring-buffer by this function. It is not necessary that the application must write the data in the context of a special API function. The write order is always linear! The first written byte is the first byte in the response message.	
Pre-conditions	
- ring-buffer has been enabled in the configuration - DescRingBufferStart() must be called before to activate the ring-buffer mechanism	
Call context	
- This API shall not interrupt the DescTask. Required for the case the currently ongoing transmission is interrupted due to a communication error, and the application still writes into the buffer.	
Particularities and Limitations	
■ `dataLength` must be lower or equal to the ring-buffer size, else the function will always fail ■ CANdesc has already filled the first bytes (SID, etc.) into the ring-buffer. So in the first call of DescRingBufferWrite() the dataLength must lower as the buffer size + these byte	

6.6.8.3 DescRingBufferCancel()

DescRingBufferCancel

Available since 5.01.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void` **`DescRingBufferCancel`** `(void)`	
Multi Context	
`void` **`DescRingBufferCancel`** `(vuint8 iContext)`	
Parameter	
`iContext`	Reference to the corresponding request context
Return code	
-	-

Functional Description

The application may call this API once the a data acquisition error has been occurred after the ring-buffer has been activated via *DescRingBufferStart()*.

CANdesc will automatically determine the appropriate action depending on its current internal state:

- if the response data transmission has not been started yet, a negative response will be sent back.
- If the response transmission has been started – a transmission interrupt will occur – the tester will not get a complete response.

Pre-conditions

- ring-buffer has been enabled in the configuration
- DescRingBufferStart() must be called before to activate the ring-buffer mechanism

Call context

-

Particularities and Limitations

■

6.6.8.4 DescRingBufferGetFreeSpace()

DescRingBufferGetFreeSpace

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`DescMsgLen` **`DescRingBufferGetFreeSpace`** `(void)`	
Multi Context	
`DescMsgLen` **`DescRingBufferGetFreeSpace`** `(vuint8 iContext)`	
Parameter	
`iContext`	reference to the corresponding request context
Return code	
`DescMsgLen`	The amount of free space/bytes in the ring-buffer.
Functional Description	
This function returns the amount of free space/bytes in the ring-buffer.	
Pre-conditions	
- ring-buffer has been enabled in the configuration - DescRingBufferStart() must be called before to activate the ring-buffer mechanism	
Call context	
-	

6.6.8.5 DescRingBufferGetProgress()

DescRingBufferGetProgress

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`DescMsgLen` **`DescRingBufferGetProgress`** `(void)`	
Multi Context	
`DescMsgLen` **`DescRingBufferGetProgress`** `(vuint8 iContext)`	
Parameter	
`iContext`	reference to the corresponding request context
Return code	
`DescRingBufferProgress`	Current byte position in the whole response.
Functional Description	
This function returns the progress of the copy process.	
Pre-conditions	
- ring-buffer has been enabled in the configuration - DescRingBufferStart() must be called before to activate the ring-buffer mechanism	
Call context	
-	
Particularities and Limitations	
■	

6.6.9 Signal Interface of CANdesc

CANdesc will provide a signal interface to the ECU application. This can help the ECU application to assemble the response automatically. No further code changes are necessary, if a signal will move or change its size.

The current implementation has only support for a synchronous signal interface. This means the ECU application has to provide the signal value within the call/context of the Signal Handler function (while reading) or to write thewithin the call/context of the Signal Handler function (while writing).

6.6.9.1 ApplDesc<Signal-Handler>()

ApplDesc<Signal-Handler>

Available since 2.00.00
Is callback ☒

Prototype	
Single Context	
- **ApplDesc<Service-Qualifier + Data-Object-Qualifier + Instance-Qualifier>** (-)	
Multi Context	
- **ApplDesc<Service-Qualifier + Data-Object-Qualifier + Instance-Qualifier>** (-)	
Parameter	
`vuint8, vsint8, vuint16, vsint16, vuint32, vsint32, DescMsg (vuint8*)`	Available for write services. Type depend on signal type
`DescMsg (vuint8*)`	Available for read services and signals > 32 bit (N bit)
Return code	
`vuint8, vsint8, vuint16, vsint16, vuint32, vsint32`	Available for read services. Type depend on signal type.
Functional Description	
A Signal Handler is generated if the Service MainHandler is configured to be generated. In this case, writing Signal Handlers are generated for all dataObjects transported with the request and reading Signal Handlers are generated for all dataObjects transported with the response (read/write from application point of view). The data type of the Signal Handler argument depends on the dataObject which is to be processed.	
Pre-conditions	
Must be configured to ‘generated’ in attribute ‘MainHandlerSupport’	
Call context	
From **DescTask()**	
Particularities and Limitations	
■ You can override the given name extension (Service-Qualifier + Data-Object-Qualifier + Instance-Qualifier) by using the SignalHandlerOverrideName.	

6.6.9.2 Configuration of direct signal access

- Application variable for direct access (default = not set)
  If this variable is specified, an access to the given external (= application) variable is generated. Nothing has to be done by the application. The external variable must be defined inside the application.
- SignalHandlerOverrideName (default = not set).
  You can adapt the name of the Signal Handler setting this value. By using this "Override Name" it is also possible to reuse an already existing Signal Handler

6.6.10 State Handling (CANdesc only)

6.6.10.1 DescGetState<StateGroup>()

DescGetState<StateGroup>

Available since 2.00.00
Is Reentrant ☒
Is callback ☐

Prototype	
Single Context	
`DescStateGroup` **`DescGetState<StateGroup-Qualifier>`** `(void)`	
Multi Context	
`DescStateGroup` **`DescGetState<StateGroup-Qualifier>`** `(void)`	
Parameter	
-	-
Return code	
`DescStateGroup`	The current state of the state group
Functional Description	

This function returns the current session state. Since the states are bit-coded the evaluation expressions may be optimized for multiple use cases.

Example: Code execution only when either the current state of this group is either state X or state Y.

```
lState = DescGetState< StateGroupQualifier >();
if ( (lState & (kDescState< StateGroupQualifier ><StateQualifier_X>) |
                kDescState< StateGroupQualifier ><StateQualifier_Y>)) != 0 )
{
 /*execute code*/
}
```

Pre-conditions
-
Call context
-
Particularities and Limitations

- For each state of a state-group a constant is defined in desc.h:
  kDescState<StateGroup-Qualifier><StateQualifier>

6.6.10.2 DescSetState<StateGroup>()

DescSetState<StateGroup>

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void` **`DescSetState<StateGroup-Qualifier>`** `(DescStateGroup newState)`	
Multi Context	
`void` **`DescSetState<StateGroup-Qualifier>`** `(DescStateGroup newState)`	
Parameter	
`DescStateGroup`	the state in which the state group should be changed
Return code	
-	-

Functional Description

By this function the state of the state-group can be changed by the ECU application. The transition notification function 'ApplDescOnTransition< StateGroupQualifier >' will be called to notify the application about the new state.

Example:

```
DescSetState<StateGroupQualifier>(kDescState<StateGroupQualifier><StateQualifier>);
```

This line will force CANdesc to change the state of the given state group to the new one.

Pre-conditions

-

Call context

-

Particularities and Limitations

- For each state of a state-group a constant will be defined in desc.h: **kDescState<StateGroup-Qualifier><State-Qualifier>**
- The **ApplDescOnTransition<StateGroup-Qualifier>()** notification function is called in any case. Also if the newState is the same as the current stat

6.6.10.3 ApplDescOnTransition«StateGroup»()

ApplDescOnTransition«StateGroup»

Available since 2.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void ApplDescOnTransition<StateGroup-Qualifier>(DescStateGroup newState, DescStateGroup formerState)`	
Multi Context	
`void ApplDescOnTransition<StateGroup-Qualifier> (DescStateGroup newState, DescStateGroup formerState)`	
Parameter	
`newState`	the CANdesc component has changed to this session state
`formerState`	the CANdesc component has changed from this session state
Return code	
-	-

Functional Description

This notification function will be called each time a transition has happened.

Pre-conditions

-

Call context

From **DescTask()**

interrupts might be disabled

Particularities and Limitations

- For each state of a state-group a constant will be defined in desc.h: **kDescState<StateGroup-Qualifier><StateName-Qualifier>**
- For some exceptions (e.g. Session) the newState can be the same as the formerState.

6.6.11 Force “Response Correctly Received - Response Pending” transmission

In some cases it is useful for the application to be sure that it has enough time to accomplish a process without causing the tester to get response timeout. In such cases the application can use the “force RCR-RP” mechanism of CANdesc, which prevents timeout between the tester and the ECU application.

How it works:

This feature is mostly applicable when a FlashBootLoader (FBL) is available for the ECU. Before starting it, the application wants to assure that there is enough time to perform reset and activate the FBL before the tester gets response timeout. The RCR-RP mechanism notifies the tester that some action is ongoing and so resets the timeout timer in the tester.

To transmit a ‘Response Correctly Received - Response Pending’ response the application has to call the DescForceRcrRpResponse() function. To be sure this response is transmitted, the application has to wait for the transmission confirmation of this forced RCR-RP response (the function ApplDescRcrRpConfirmation). Depending on its transmission status parameter the application can decide how the processing shall continue (a jump to FBL or to close the request processingth negative response).

6.6.11.1 DescForceRcrRpResponse()

DescForceRcrRpResponse

Available since 2.11.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void` **`DescForceRcrRpResponse`**`(void)`	
Multi Context	
`void` **`DescForceRcrRpResponse`**`(vuint8 iContext)`	
Parameter	
`iContext`	reference to the corresponding request context
Return code	
-	-
Functional Description	
Calling this function the application can force CANdesc to send immediately (not later than the next call of DescTask() function) a RCR-RP response.	
Pre-conditions	
CANdesc was configured to use this option (enabled in the GENtool).	
Call context	
Task or interrupt.	
Particularities and Limitations	
■ This function can be called: after a call of a MainHandler function (e.g. **ApplDescCheckSessionTransition()**) and until the call of **ApplDescResponsePendingOverrun()** or **ApplDescResponsePendingOvertimed()** or**pConfirmation().**	

6.6.11.2 ApplDescRcrRpConfirmation()

ApplDescRcrRpConfirmation

Available since 2.11.00
Is callback ☒

Prototype	
Single Context	
`void ApplDescRcrRpConfirmation(vuint8 status)`	
Multi Context	
`void ApplDescRcrRpConfirmation(vuint8 iContext, vuint8 status)`	
Parameter	
`iContext`	Reference to the corresponding request context
`status`	If the transmission was successful, the parameter value will be *kDescOk*. Otherwise – *kDescFailed.*
Return code	
-	-
Functional Description	
Once the RCR-RP response has been forced, this function will be called in any case. The transmission status is reported by the status parameter.	
Pre-conditions	
CANdesc was configured to use this option (enabled in the GENtool).	
Call context	
CAN Driver TX-ISR → TP Confirmation → this function	
Particularities and Limitations	
■ Be aware of time consuming implementation for this function (interrupt call context).	

6.6.12 DynamicallyDefineDataIdentifier ($2C) (UDS) functions

Since this feature is only for some OEM available, please refer to the OEM specific documentation to find out if is applicable for your configuration.

6.6.12.1 DescMayCallStateTaskAgain()

DescMayCallStateTaskAgain

Available since 4.00.00
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`DescBool` **`DescMayCallStateTaskAgain`** `(void)`	
Multi Context	
`DescBool` **`DescMayCallStateTaskAgain`** `(void)`	
Parameter	
-	-
Return code	
`kDescTrue`	TRUE if you may call again the state task within this application task cycle.
`kDescFalse`	FALSE if the DescStateTask() must not be called again.

Functional Description

Motivation: The ***DescStateTask()*** can be called as fast as possible but it still can not be enough fast for complex service processing (e.g. DDIDs containing long descriptions) to match fast timing-performance requirements. This function provides the info if the application may call again the state-task in the same task context without causing endless loop (important for non-preemptive OS environments).

Example of the API usage:

```
void ApplDiagTask(void) /* application function called as fast as possible */
{
   do /* pump the state task as long as needed */
   {
     DescStateTask();
   }
   while(DescMayCallStateTaskAgain() == kDescTrue);
}
```

Pre-conditions

- Preprocessor define “**DESC_ENABLE_HIPERFORMANCE_DYNDID_MODE**” is available (using user-config file in GENtool).

- The application uses the split-task concept (i.e. calls ***DescState-/TimerTask()*** instead of ***DescTask()***).

Call context

Background-loop level or OSEK-OS Task. The Task should have a lower or equal priority than all other interaction to the CANdesc component.

Particularities and Limitations

■

6.6.12.2 ApplDescCheckDynDidMemoryArea()

ApplDescCheckDynDidMemoryArea
Available since 3.02.00
Must be Reentrant ☒
Is callback ☒

Prototype

Any Context

```
DescDynDidMemCheckResult  ApplDescCheckDynDidMemoryArea (
 DescDynDidMemBlockAddress srcAddr,
 DescDynDidMemBlockSize    len      );
```

Parameter

`srcAddr`	Start address (Service $2C 02 request parameter 'memoryAddress').
`len`	Length of block to read (Service $2C 02 request parameter 'memorySize').

Return code

`memBlockOk`	Permit the access to requested memory block and extend the DDID.
`memBlockInvAddress`	Forbid the access due invalid requested memory address (requestOutOfRange).
`memBlockInvSize`	Forbid the access due invalid requested block length (requestOutOfRange).
`memBlockInvSecurity`	Forbid the access due current security mode settings prohibit the DDID definition (securityAccessDenied).
`memBlockInvCondition`	Forbid the access due other restrictions (conditionsNotCorrect).

If the memory access if forbidden, the Service $2C Request is negative responded with NRC 22 (conditionsNotCorrect), 31 (requestOutOfRange) or 33 (securityAccessDenied).

Functional Description

This callback function is triggered when defining a DDID that shall read bytes from the ECU's memory (Service Request $2C 02). The application can permit the (re-)definition of the DDID or forbid it.
The service request is responded according to this.

The application must check

- if the given `srcAddr` and following `len` bytes are valid ECU addresses and if they are readable,
- if the current security state allows to define the DDID right now,
- if there are other conditions that may forbid the definition of the DDID.

If all checks allow the DDID definition, the callback function must return `memBlockOk`.

FYI: When later reading the defined DDIDs by service $22, the standard checks [of Service $23 ReadMemoryByAddress] are executed, that perform security checks before accessing the memory.
So, above security check with service $2C shall prove that the current security state permits the *definition* of the DDID, the security check in service $22 (resp. $23) proves [in the context of the then existing security state] the actual *reading* of the memory range.

Pre-conditions

-
Call context
From **DescTask()**
Particularities and Limitations
•

6.6.12.3 Non-volatile memory support

For some car-manufactures CANdesc provides NVRAM support for the dynamically defined DID definitions. There are some APIs that must be operated and some call-backs to be implemented by the application in order to get the NVRAM support fully operational.

The following diagrams show the two oeprations on NVRAM – restore (at power on) and st ore (usuall prior power off) data.

Restore data at ECU power on

	**Caution** At each CANdesc initialization (e.g. ECU reset/ power on) the “restore” procedure must be performed!

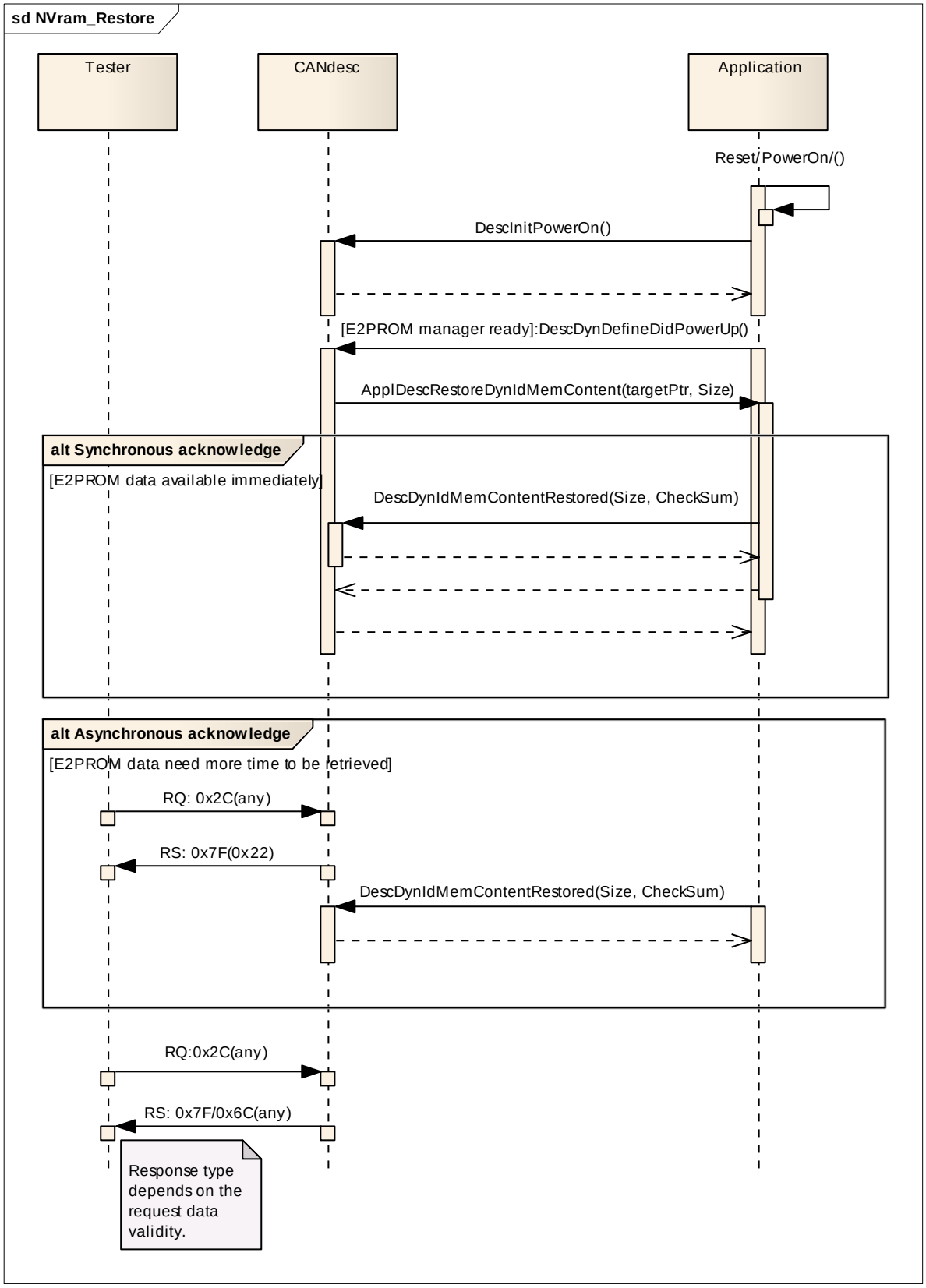


Figure 6-1 DynDID definition restore and tester interaction

Store data at ECU power down

Info
The store operation can be performed at any time not only at power down.

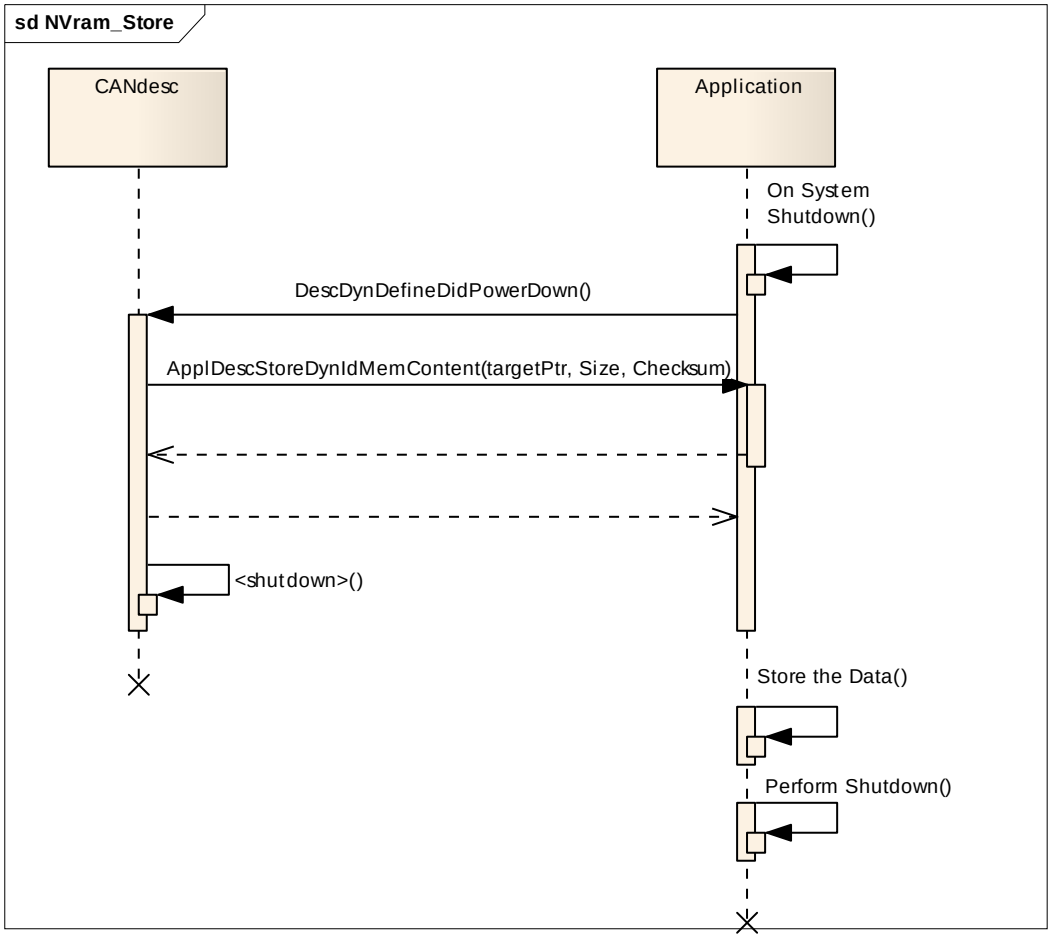


Figure 6-2 Store DynDID definitions

6.6.12.3.1 DescDynDefineDidPowerUp()

DescDynDefineDidPowerUp
Available since 5.06.09
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void` **`DescDynDefineDidPowerUp`** `(void)`	
Multi Context	
`void` **`DescDynDefineDidPowerUp`** `(void)`	
Parameter	
-	-
Return code	
-	-
Functional Description	
Once the ECU has been powered one/reset or just need to be reinitialized, this API must be called to restore the dynamically defined DID content. Usually called after the NVRAM manager is initialized.	
Pre-conditions	
- Service 0x2C needs to store the DynDID definitions to the NVRAM (OEM specific requirement)	
Call context	
- any	
Particularities and Limitations	
■ Must be called after DescInitPowerOn().	

6.6.12.3.2 DescDynIdMemContentRestored ()

DescDynIdMemContentRestored
Available since 5.06.09
Is Reentrant ☐
Is callback ☐

Prototype	
Single Context	
`void` **`DescDynIdMemContentRestored`** `(DescDynDidStorageInfo storageInfo)`	
Multi Context	
`void` **`DescDynIdMemContentRestored`** `(DescDynDidStorageInfo storageInfo)`	
Parameter	
`storageInfo.nvData`	Not used
`storageInfo.nvDataSize`	The size (in bytes) of the restored table.
`storageInfo.checkSum`	The stored checksum, calculated by CANdesc at store time.
Return code	
-	-
Functional Description	
After CANdesc has requested the application to restore the DynDID data ("*ApplDescRestoreDynIdMemContent ()*"), this API must be called to notify CANdesc that the DynDID content has been restored and can be used.	
Pre-conditions	
- Service 0x2C needs to store the DynDID definitions to the NVRAM (OEM specific requirement)	
Call context	
- any	
Particularities and Limitations	
■ none	

6.6.12.3.3 DescDynDefineDidPowerDown ()

DescDynDefineDidPowerDown
Available since 5.06.09
Is Reentrant ☐
Is callback ☐

Prototype
Single Context
`void` **`DescDynDefineDidPowerDown`** `(void)`
Multi Context
`void` **`DescDynDefineDidPowerDown`** `(void)`

Parameter	
-	-

Return code	
-	-

Functional Description

If the ECU has to be reset or just power off /shutdown, this API must be called to store the current DID definitions.

In order to save E2PROM write cycles, the application may perform compare to the current E2PROM content and decide whether to store the table content or not.

Pre-conditions

- Service 0x2C needs to store the DynDID definitions to the NVRAM (OEM specific requirement)

Call context

- any

Particularities and Limitations

- Shall be called prior power-down/shutdown execution
- May be called any time to store the current content of the DynDID tables.

6.6.12.3.4 ApplDescStoreDynIdMemContent ()

ApplDescStoreDynIdMemContent
Available since 5.06.09
Is Reentrant ☐
Is callback ☒

Prototype	
Single Context	
`void ApplDescStoreDynIdMemContent (DescDynDidStorageInfo storageInfo)`	
Multi Context	
`void ApplDescStoreDynIdMemContent (DescDynDidStorageInfo storageInfo)`	
Parameter	

`storageInfo.nvData`
`storageInfo.nvDataSize`
`storageInfo.checkSum`
	The pointer to the data to be stored;
	The size (in bytes) of the table;
	The checksum value, calculated by CANdesc, to be stored.

Return code	
-	-
Functional Description	
Once this API is called by CANdesc, the application must trigger a write E2PROM procedure to store the data given by CANdesc and the checksum value. In order to save E2PROM write cycles, the application may perform compare to the current E2PROM content and decide whether to store the table content or not.	
Pre-conditions	
- Service 0x2C needs to store the DynDID definitions to the NVRAM (OEM specific requirement)	
Call context	
- any	
Particularities and Limitations	
■ CANdesc does not keep the data pointed by the parameter pointer during the write operation! The application must mirror the data if needed!	

6.6.12.3.5 ApplDescRestoreDynIdMemContent ()

ApplDescRestoreDynIdMemContent
Available since 5.06.09
Is Reentrant ☐
Is callback ☒

Prototype	
Single Context	
`void ApplDescRestoreDynIdMemContent (DescDynDidStorageInfo storageInfo)`	
Multi Context	
`void ApplDescRestoreDynIdMemContent (DescDynDidStorageInfo storageInfo)`	
Parameter	
`storageInfo.nvData`	The pointer to the data to where the stored data shall be written
`storageInfo.nvDataSize`	The size (in bytes) of the table expected.
`storageInfo.checkSum`	Not used
Return code	
-	-
Functional Description	
Once this API is called by CANdesc, the application must trigger a read E2PROM procedure to restore the data for CANdesc and the checksum value. Once the read process has completed, the API "*DescDynIdMemContentRestored ()*" must be called to acknowledge the operation status to CANdesc.	
Pre-conditions	
- Service 0x2C needs to store the DynDID definitions to the NVRAM (OEM specific requirement)	
Call context	
- any	
Particularities and Limitations	
■	

6.6.13 Memory Access Callbacks

6.6.13.1 ApplDescReadMemoryByAddress()

ApplDescReadMemoryByAddress

Available since 5.06.04
Is Reentrant ☐
Is callback ☒

Prototype	
Any Context	
`void ApplDescReadMemoryByAddress (DescMsgContext* pMsgContext, t_descMemByAddrInfo* pMemInfo)`	
Parameter	
`pMsgContext`	Refer the section 6.6.3.2 Service MainHandler for details about this parameter.
`pMsgContext->resData`	The response buffer pointer
`pMsgContext->resDataLen`	The actual response length
`pMemInfo->address`	The address to read from
`pMemInfo->length`	The number of bytes to read
Return code	
-	-

Functional Description

This callback is called for read memory by address requests. The application has to do following:

- Perform memory block validation (negative response can be set by calling *DescSetNegResponse()*).
- Optional: Perform additional state validations (negative response can be set by calling *DescSetNegResponse()*).
- Copy the requested memory contents into the response buffer.
- Set the response data length to the number of bytes copied.
- Confirm that the processing is finished (by calling *DescProcessingDone()*).

Pre-conditions

- The read memory by address service is supported.
- Refer to chapter *5.5Read/Write Memory by Address (SID $23/$3D) (UDS)* for more details of the availability of this API. If you don't see this API provided in desc.h, then this feature is not supported for your project.

Call context

From **DescTask()**

Particularities and Limitations

- To call this handler periodically, 'DescStartMemByAddrRepeatedCall' needs to be used

6.6.13.2 ApplDescWriteMemoryByAddress()

ApplDescWriteMemoryByAddress

Available since 5.06.04
Is Reentrant ☐
Is callback ☒

Prototype	
Any Context	
`void ApplDescWriteMemoryByAddress (DescMsgContext* pMsgContext, t_descMemByAddrInfo* pMemInfo)`	
Parameter	
`pMsgContext`	Refer the section 6.6.3.2 Service MainHandler for details about this parameter.
`pMsgContext->reqData`	The pointer to the data to store
`pMemInfo->address`	The address to write to
`pMemInfo->length`	The number of bytes to write
Return code	
-	-

Functional Description

This callback is called for write memory by address requests. The application has to do following:

- Perform memory block validation (negative response can be set by calling *DescSetNegResponse()*).
- Optional: Perform additional state validations (negative response can be set by calling *DescSetNegResponse()*).
- Copy the provided data into the memory area.
- Confirm that the processing is finished (by calling *DescProcessingDone()*).

Pre-conditions

- The write memory by address service is supported.
- Refer to chapter *5.5Read/Write Memory by Address (SID $23/$3D) (UDS)* for more details of the availability of this API. If you don't see this API provided in desc.h, then this feature is not supported for your project.

Call context

From **DescTask()**

Particularities and Limitations

- To call this handler periodically, 'DescStartMemByAddrRepeatedCall' needs to be used

6.6.14 Flash Boot Loader Support

CANdesc provides some features to comply with the HIS flash boot loader procedures.

These features are not released for all OEMs so if the below listed APIs are not available in your CANdesc version, then for the OEM, you currently use CANdesc, does not require, resp. has another FBL procedures.

6.6.14.1 DescSendPosRespFBL()

DescSendPosRespFBL

Available since 4.05.00
Is Reentrant ☐
Is callback ☐

Prototype	
Any Context	
`void` **`DescSendPosRespFBL`** `(t_descFblPosRespType posRespSId)`	
Parameter	
`posRespSId`	One of the following values are allowed: ■ kDescSendFblPosRespEcuHardReset ■ kDescSendFblPosRespDscDefault.
Return code	
-	-
Functional Description	
The application shall call this function as soon as possible after the initialization of the CANdesc component is done and the ECU is able to communicate. Once this function called, CANdesc will try to send the corresponding positive response as follows: ■ kDescSendFblPosRespEcuHardReset – a positive response to EcuHardReset ($51 $01) will be sent. ■ kDescSendFblPosRespDscDefault – a positive response to DiagnosticSessionControl Default session ($50 $01 $P2time $P2Star/10) will be sent. If CANdesc is currently busy with a new tester request, there will be no response sent by this API.	
Pre-conditions	
The FBL positive response feature is supported.	
Call context	
Any.	
Particularities and Limitations	
■ See 7.8	

6.6.14.2 ApplDescInitPosResFblBusInfo()

ApplDescInitPosResFblBusInfo
Available since 5.07.04
Is Reentrant ☐
Is callback ☒

Prototype	
Any Context	
`vuint8 ApplDescInitPosResFblBusInfo (t_descUsdtNetBus* pBusInfo)`	
Parameter	
`pBusInfo`	Reference to the bus information structure that will be initialized here.
`pBusInfo->busType`	The bus driver that will send the response
`pBusInfo->comChannel`	The communication channel on which the response will be sent. (relevant only on multi channel systems)
`pBusInfo->testerId`	The tester address which will be respond to. (relevant only on bus systems with source/target addresses)
Return code	
`kDescOk`	Operation was successful, the FBL positive response will be sent.
`kDescFailed`	Operation failed – no FBL positive response will be sent.

Functional Description

This callback is called once the application decided to call the API *DescSendPosRespFBL* to get the concrete addressing information.

The application shall initialize only the parameter described above. The optional ones can be skipped if not relevant on your system.

Pre-conditions

The FBL positive response feature is supported.

Call context

From *DescSendPosRespFBL* context.

Particularities and Limitations

- -

6.6.15 Debug Interface / Assertion

6.6.15.1 ApplDescFatalError()

ApplDescFatalError
Available since 2.00.00
Is Reentrant ☐
Is callback ☒

Prototype	
Single Context	
`void ApplDescFatalError (vuint8 errorCode, vuint16 lineNumber)`	
Multi Context	
`void ApplDescFatalError (vuint8 errorCode, vuint16 lineNumber)`	
Parameter	
`errorCode`	The errorCode is a classification of the assertion. The errorCodes can be also found in file ‘desc.h’. The errorCodes are listed below:
`lineNumber`	A line number of file ‘desc.c’ from which this function is called.
Return code	
-	-

Functional Description

The CANdesc debug interface is similar to assertion constructof common programming languages. Assertions are code checks which are written so that they should always evaluate to true. If an assertion is false, it indicates a possible bug in the program, corrupt system state or a misoperation of the user-interface.

CANdesc is calling the function ApplDescFatalError() function to indicate a evaluation of an assertion to false. If this will happen it is recommended to halt the program's execution immediately. This could be reach by an endless loop in that call-back.

The assertions can be disabled in the GenTool settings. The resource (ROM and runtime) consumption can be reduced by disabling the assertions.

Error codes

kDescAssertWrongTpTxChannel (0x00):

The wrong TP channel is used – verify the TP interface to the CANdesc component

kDescAssertIndexTableInvalidReference (0x02):

Internal generation failure.

kDescAssertSvcTableUnreachableItem (0x03):

Internal generation failure.

kDescAssertSvcTableInvalidReference (0x04):

Internal generation failure.

kDescAssertSvcTableInconsistentNumber (0x05):
Internal generation failure.

kDescAssertMissingMainHandler (0x06):
Internal generation failure.

kDescAssertInvalidContextId (0x08):
Wrong iContext should be used - Check the consistency of the iContext parameter in the application.

kDescAssertSvcTableIndexOutOfRange (0x09):
Internal generation failure.

kDescAssertSvcInstTableIndexOutOfRange (0x0A):
Internal generation failure.

kDescAssertContextIdWasModified (0x0B):
The iContext member of the pMsgContext parameter in the MainHandler functions are illegal modified – verify the MainHandler functions in the application

kDescAssertProcessingDoneCallAfterResFlushing (0x0E):
DescProcessingDone() is called at least twice for one request – check the call of DescProcessingDone() in the application.

kDescAssertTooLongSingleFrameResponse (0x0F):
Response lengthof a periodic DID is exceeding the SingleFrame length – check the response length for periodic DIDs.

kDescAssertApplLackOfConfirmation (0x11):
The time for response processing is too long – verify if the call of DescProcessingDone() is done in any case.

kDescAssertZeroStateValue (0x13):
The state parameter is zero – check state handling

kDescAssertInvalidContextMode (0x16):
Internal runtime error

kDescAssertUnexpectedWriteIntoRingBuffer (0x17):
DescRingBufferWrite() is called without activated ring-buffer

kDescAssertRingBufferWriteExceedsTheResLen (0x18):
DescRingBufferWrite() is called to often

kDescAssertIllegalUsageOfNegativeResponse (0x1A):
After call of DescProcessingDone() a negative response is set

kDescAssertDiagnosticBufferOverflow (0x1B):
currently not available

kDescAssertFuncReqWoResMayNotUseRingBuffer (0x1C):
It is not possible to use the ring-buffer feature for functional request (KWP only)

kDescAssertSchedulerTimerEventWithoutAnyPID (0x1E):
Internal runtime error

kDescAssertSchedulerRingBufferIsActivated (0x1F):
For periodic DIDs it is not possible to use the ring-buffer.

kDescAssertUnknownTpTransmissionType (0x21):
Internal runtime error

kDescAssertIllegalAddRequestCount (0x22):
Internal runtime error

kDescAssertNoSidCanBeReportedInIdleMode (0x23):
Call of DescGetSeriveId() while not a user-service is processed

kDescAssertInvalidUsageOfForceRcrRpApi (0x24):
The DescForceRcrRpResponse() function is used illegal.

kDescAssertPidResLenToCddDefNotMatched (0x26):
The response length set by the application do not fit to the response length defined in CANdela (cdd).

kDescAssertPidResLenToCurrLinearFreeSpace (0x27):
Internal runtime error

kDescAssertMissingDataForTransmission (0x28):
Internal runtime error

kDescAssertSchedulerFreeCellNotFound (0x29):
Internal runtime error

kDescAssertInvalidStateParameterValue (0x2A):
The state parameter value is wrong – check state handling in your application

kDescAssertNoFreeICNChannel (0x2B):
Internal runtime error

kDescAssertInvalidDescICNClient (0x2C):
Internal runtime error

kDescAssertNoFreeMsgContext (0x2D):
Internal runtime error

kDescAssertUnExpectedContextWithResponse (0x2E):
A response will be sent out of a wrong context.

kDescAssertIllegalCallOfRingBufferCancel (0x2F):
The API *DescRingBufferCancel()* has been called for a response that is not using the ring-buffer concept (e.g. *DescRingBufferStart()* was not called).

kDescNetAssertWrongIsoTpRxChannel (0x40):
The wrong TP channel is used – verify the TP interface to the CANdesc component

kDescNetAssertWrongIsoTpTxChannel (0x41):
The wrong TP channel is used – verify the TP interface to the CANdesc component

kDescNetAssertWrongBusType (0x42):
The wrong bus type is used – verify the TP interface to the CANdesc component

kDescAssertDescIcnIllegalTargetPointer (0x50):
Internal runtime assertion

Pre-conditions

At least on type of assertions are activated

Call context

Form ISR or task level. The interrupts might be disabled

Particularities and Limitations

- After a call of this function the system is not stable anymore. It can not be guaranteed that this component or the whole system is still working in correct manner.

7 How To…

7.1 …implement a protocol service MainHandler

```
//1. Read ProtocolService
// - dynamic length
// - PIDs

void DESC_API_CALLBACK_TYPE ApplDescManiOnTimerEvent_storeEvent(DescMsgContext*
pMsgContext)
{
  /* Check the length */
  if(pMsgContext->reqDataLen > 2)
  {
    /* Check the sub-parameters */
    vuint16 param;
    /* Compose one parameter combining the HiByte and the LoByte in this order*/
    param = DescMake16Bit(pMsgContext->reqData[0], pMsgContext->reqData[1]);

    /* Dispatch the parameter */
    switch(param)
    {
      case 0xFFFF:
        if(pMsgContext->reqDataLen != 0xFFFF)
        {
          /* Write some data (skip the parameter offsets 0 und 1) */
          pMsgContext->resData[2] = DescGetLoByte(0x1234);
          pMsgContext->resData[3] = DescGetHiByte(0x1234);
          /* Set the response length */
          pMsgContext->resDataLen = 4;
        }
        else
        {
          DescSetNegResponse(pMsgContext->iContext, kDescNrcInvalidFormat);
        }
        break;
      default:
        /* unknown parameter */
        DescSetNegResponse(pMsgContext->iContext, kDescNrcInvalidFormat);
    }
  }
  else
  {
    DescSetNegResponse(pMsgContext-iContext, kDescNrcInvalidFormat);
  }
  /* In this case we did everything in the main-handler */
  DescProcessingDone(pMsgContext->iContext);
}


//2. Read ProtocolService
// - dynamic length
// - sub-function

void DESC_API_CALLBACK_TYPE ApplDescManiOnTimerEvent_storeEvent(DescMsgContext*
```

```
pMsgContext)
{
  /* Check the length */
  if(pMsgContext->reqDataLen > 1)
  {
    /* Dispatch the sub-function */
    switch(pMsgContext->reqData[0])
    {
      case 0xFF:
        if(pMsgContext->reqDataLen != 0xFFFF)
        {
          /* Format check ok: write some data (skip the parameter) */
          pMsgContext->resData[1] = DescGetLoByte(0x1234);
          pMsgContext->resData[2] = DescGetHiByte(0x1234);
          /* Set the response length */
          /* Hint: if the response length wasn't set, zero value is assumed! */
          pMsgContext->resDataLen = 3;
        }
        else
        {
          /* Wrong sub-parameter format */
          DescSetNegResponse(pMsgContext->iContext, kDescNrcInvalidFormat);
        }
        break;
      default:
        /* Unknown sub-function */
        DescSetNegResponse(pMsgContext->iContext,
                           kDescNrcSubfunctionNotSupported);
    }
  }
  else
  {
    DescSetNegResponse(pMsgContext-iContext, kDescNrcInvalidFormat);
  }
  /* In this case we did everything in the main-handler */
  DescProcessingDone(pMsgContext->iContext);
}

//3. Write ProtocolService
// - dynamic length
// - PIDs

void DESC_API_CALLBACK_TYPE ApplDescManiOnTimerEvent_storeEvent(DescMsgContext*
pMsgContext)
{
  /* Check the sub-parameters */
  vuint16 param;

  /* Check the length */
  if(pMsgContext->reqDataLen > 2)
  {
    /* Compose one parameter combining the HiByte and the LoByte in this order
*/
    param = DescMake16Bit(pMsgContext->reqData[0], pMsgContext->reqData[1]);

    /* Dispatch the parameter */
    switch(param)
    {
      case 0xFFFF:
        if(pMsgContext->reqDataLen != 0xFFFF)
```

```
        {
          /* Copy from the request data to your application */
          /* Use the data pointed by: pMsgContext->reqData[2],
             pMsgContext->reqData[3], etc.*/
        }
        else
        {
          DescSetNegResponse(pMsgContext->iContext, kDescNrcInvalidFormat);
        }
        break;
      default:
        /* unknown parameter */
        DescSetNegResponse(pMsgContext->iContext, kDescNrcRequestOutOfRange);
    }
  }
  else
  {
    DescSetNegResponse(pMsgContext-iContext, kDescNrcInvalidFormat);
  }
  /* In this case we did everything in the main-handler */
  /* Hint: if the response length wasn't set, zero value is assumed! */
  DescProcessingDone(pMsgContext->iContext);
}

//4. Write ProtocolService
// - dynamic length
// - Sub-function

void DESC_API_CALLBACK_TYPE ApplDescManiOnTimerEvent_storeEvent(DescMsgContext*
pMsgContext)
{
  /* Check the sub-parameters */
  vuint16 param;

  /* Check the length */
  if(pMsgContext->reqDataLen > 2)
  {
    /* Compose one parameter combining the HiByte and the LoByte in this order*/
    param = DescMake16Bit(pMsgContext->reqData[0], pMsgContext->reqData[1]);

    /* Dispatch the parameter */
    switch(param)
    {
      case 0xFFFF:
        if(pMsgContext->reqDataLen != 0xFFFF)
        {
          /* Copy from the request data to your application */
          /* Use the data pointed by: pMsgContext->reqData[2],
             pMsgContext->reqData[3], etc.*/
        }
        else
        {
          DescSetNegResponse(pMsgContext->iContext, kDescNrcInvalidFormat);
        }
        break;
      default:
        /* unknown sub-function /
        DescSetNegResponse(pMsgContext->iContext,
                           kDescNrcSubfunctionNotSupported);
    }
```

```
  }
  else
  {
    DescSetNegResponse(pMsgContext-iContext, kDescNrcInvalidFormat);
  }
  /* In this case we did everything in the main-handler */
  /* Hint: if the response length wasn't set, zero value is assumed! */
  DescProcessingDone(pMsgContext->iContext);
}
```

7.2 …implement a service MainHandler

```
//5. Read Service
// - dynamic length
// - sub-function/PID

void DESC_API_CALLBACK_TYPE ApplDescManiOnTimerEvent_storeEvent(DescMsgContext*
pMsgContext)
{
  /* Check the length */
  if(pMsgContext->reqDataLen != 0xFFFF)
  {
    /* Format check ok: write some data */
    pMsgContext->resData[0] = DescGetLoByte(0x1234);
    pMsgContext->resData[1] = DescGetHiByte(0x1234);
    /* Set the response length */
    /* Hint: if the response length wasn't set, zero value is assumed! */
    pMsgContext->resDataLen = 2;
  }
  else
  {
    /* Wrong sub-function format */
    DescSetNegResponse(pMsgContext->iContext, kDescNrcInvalidFormat);
  }

  /* In this case we did everything in the main-handler */
  DescProcessingDone(pMsgContext->iContext);
}


//6. Read Service
// - static length
// - sub-function/PID

void DESC_API_CALLBACK_TYPE ApplDescManiOnTimerEvent_storeEvent(DescMsgContext*
pMsgContext)
{
  /* Format check ok: write some data */
  pMsgContext->resData[0] = DescGetLoByte(0x1234);
  pMsgContext->resData[1] = DescGetHiByte(0x1234);
  /* Set the response length */
  /* Hint: if the response length wasn't set, zero value is assumed! */
  pMsgContext->resDataLen = 2;

  /* In this case we did everything in the main-handler */
  DescProcessingDone(pMsgContext->iContext);
}
```

```
//7. Write Service
// - dynamic length
// - sub-function/PID

void DESC_API_CALLBACK_TYPE ApplDescManiOnTimerEvent_storeEvent(DescMsgContext*
pMsgContext)
{
  /* Check the length */
  if(pMsgContext->reqDataLen != 0xFFFF)
  {
    /* Format check ok: write some data */
    /* Copy from the request data to your application */
    /* Use the data pointed by: pMsgContext->reqData[0],
       pMsgContext->reqData[1], etc.*/
  }
  else
  {
    /* Wrong sub-function format */
    DescSetNegResponse(pMsgContext->iContext, kDescNrcInvalidFormat);
  }

  /* In this case we did everything in the main-handler */
  /* Hint: if the response length wasn't set, zero value is assumed! */
  DescProcessingDone(pMsgContext->iContext);
}


//8. Write  Service
// - static length
// - sub-function/PID

void DESC_API_CALLBACK_TYPE ApplDescManiOnTimerEvent_storeEvent(DescMsgContext*
pMsgContext)
{
  /* Copy from the request data to your application */
  /* Use the data pointed by: pMsgContext->reqData[0], pMsgContext->reqData[1],
     etc.*/

  /* In this case we did everything in the main-handler */
  /* Hint: if the response length wasn't set, zero value is assumed! */
  DescProcessingDone(pMsgContext->iContext);
}
```

7.3 …implement a Signal Handler

```
//1. ReadSignalHandler
// - length <= 4Byte
// Limitations: No DescProcessingDone() or DescSetNegResponse() allowed.

vuintx DESC_API_CALLBACK_TYPE ApplDescGetTemp(void)
{
  /* Return directly the signal value */
  return (vuintx)0xFFFF;
}


//2. ReadSignalHandler
```

```
// - length > 4Byte
// Limitations: No DescProcessingDone() or DescSetNegResponse() allowed.

DescMsgLen DESC_API_CALLBACK_TYPE ApplDescGetTemp(DescMsg tgt)
{
  /* Copy the signal data into the buffer pointed by "tgt".*/
  /* Return the amount of written bytes */
  return 0;
}


//3. WriteSignalHandler
// - length <= 4Byte
// Limitations: No DescProcessingDone() or DescSetNegResponse() allowed.

void DESC_API_CALLBACK_TYPE ApplDescGetTemp(vuintx data)
{
  /* "data" contains the signal value as-is from the request.
     Copy it into your application. */
}


//4. ReadSignalHandler
// - length > 4Byte
// Limitations: No DescProcessingDone() or DescSetNegResponse() allowed.

DescMsgLen DESC_API_CALLBACK_TYPE ApplDescGetTemp(DescMsg src)
{
  /* Copy the signal data from the buffer pointed by "src".*/
  /* Return the amount of copied bytes */
  return 0;
}
```

7.4 …implement a Packet Handler

```
//1. ReadPacketHandler
// Limitations: No DescProcessingDone() or DescSetNegResponse() allowed.

void DESC_API_CALLBACK_TYPE ApplDescGetTemp(DescMsg pMsg)
{
  /* Copy the signal value into the "pMsg" buffer. */
  pMsg[0] = DescGetLoByte(0x1234);
  pMsg[1] = DescGetLoByte(0x1234);
}
```

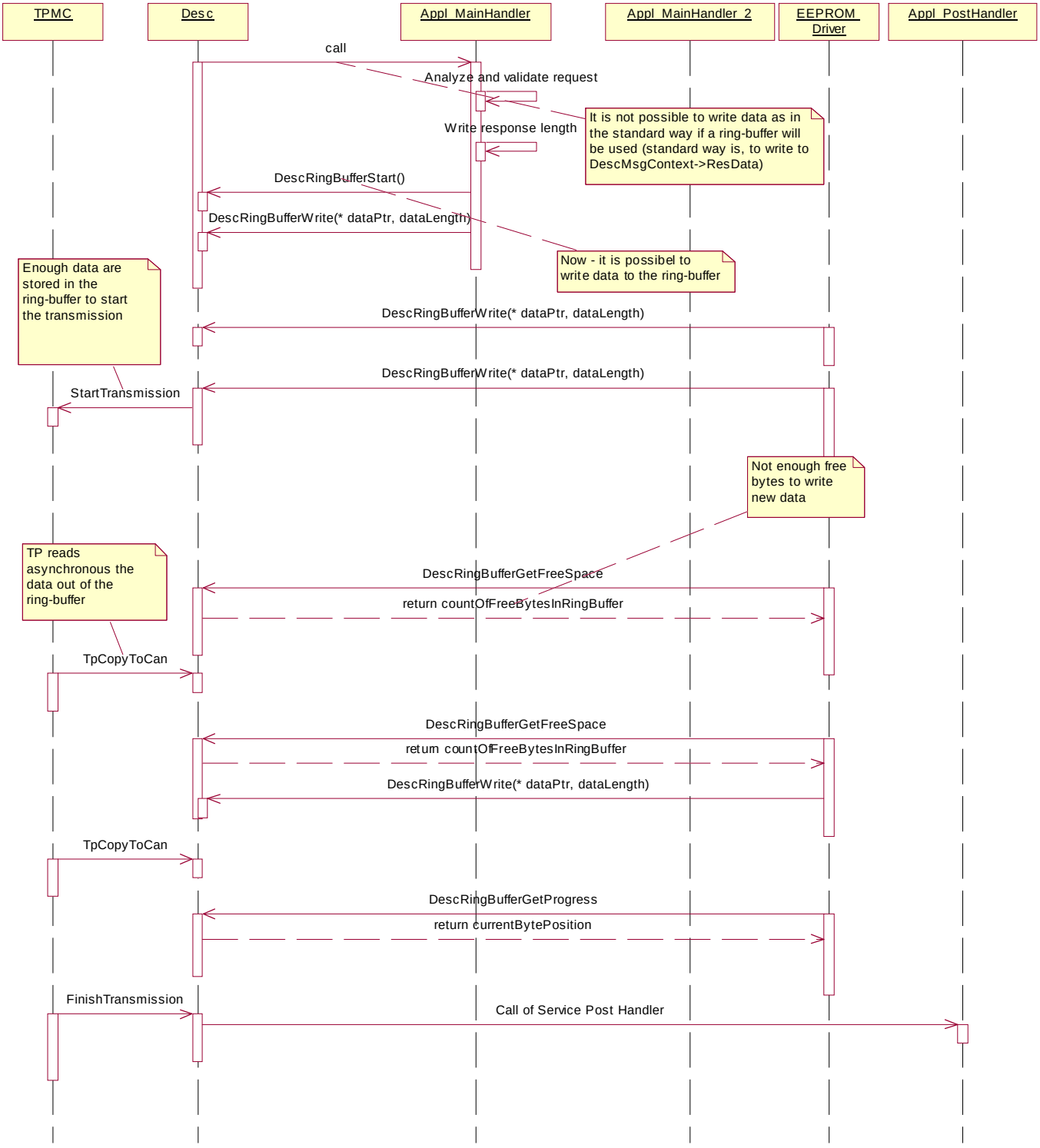
7.5 …implement a state transition function

```
//1. StateTransitionNotification
// Limitations: No DescProcessingDone() or DescSetNegResponse() allowed.

void DESC_API_CALLBACK_TYPE ApplDescOnTransitionSession(DescStateGroup
formerState, DescStateGroup newState)
{
  /* You are just notified that this state group has performed a transition from
   * "formerState" to the "newState". */
```

```
}
```

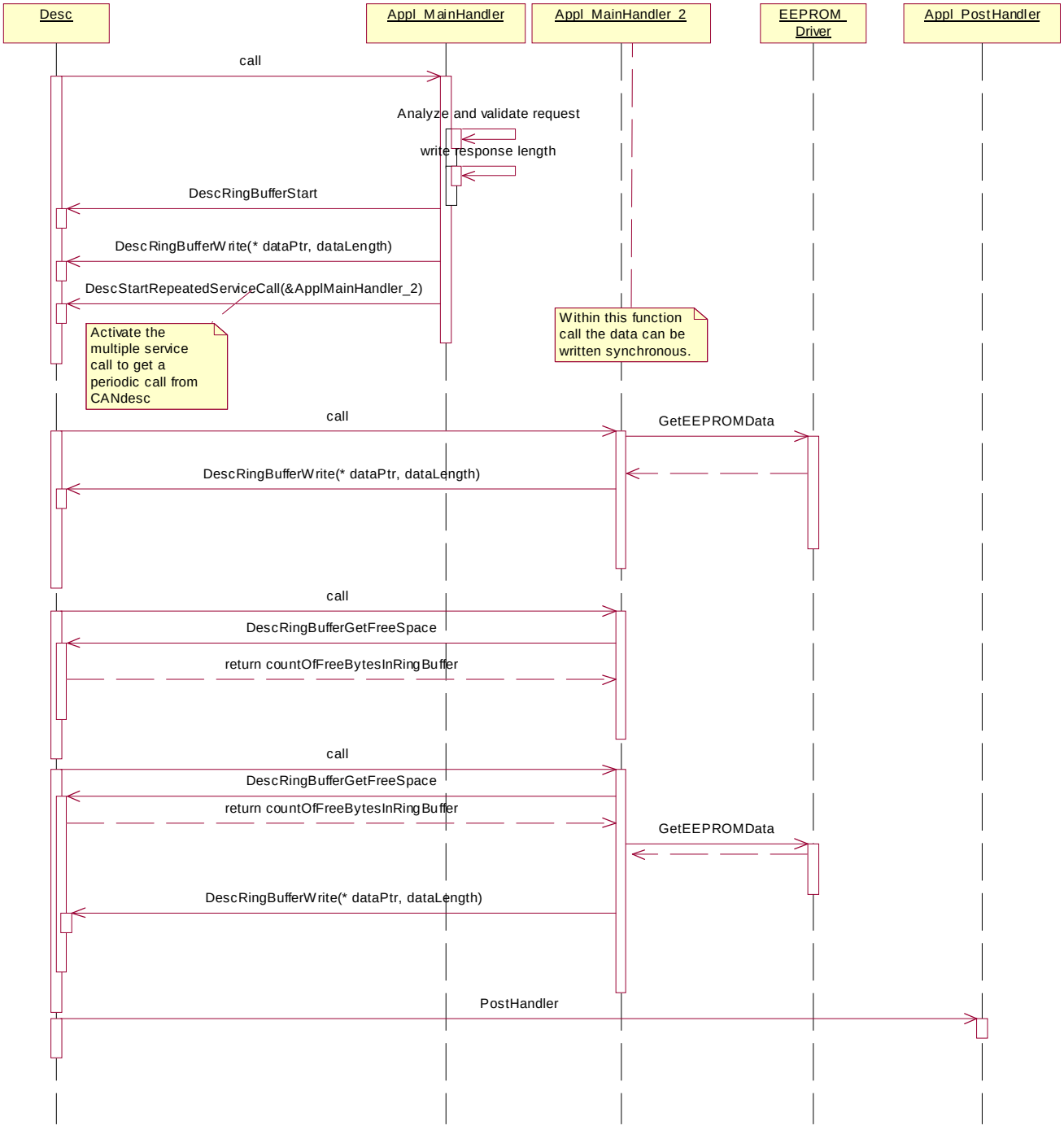
7.6 …work with the ring-buffer mechanism

7.6.1 with asynchronous write



```
//1. Read Service (with asynchronous Ring-Buffer)
// - static length
// - sub-function/PID

vuint8 g_iContext;

void DESC_API_CALLBACK_TYPE ApplDescReadDTC(DescMsgContext* pMsgContext)
{
  vuint8 lData;
  /* Format check already done by CANdesc */

  /* Analysis of request has to done by ECU application */

  /* Set the response length */
  pMsgContext->resDataLen = 16;

  /* Fill the first data */
  lData = 5;

  /* Store iContext for further interaction with CANdesc */
  g_iContext = pMsgContext->iContext;
  /* check only on services with sub-function (e.g. 0x19) */
  if(pMsgContext->msgAddInfo.suppPosRes != 0)
  {
    /* since no response required – skip further processing */
    DescProcessingDone(pMsgContext->iContext);
  }
  else
  {
   /* Now we have to set CANdesc into the Ring-Buffer mode */
   DescRingBufferStart(pMsgContext->iContext);
   /* Now it is possible to write into the Ring-Buffer */
   DescRingBufferWrite(pMsgContext->iContext, &lData, 1);

   /* Now trigger e.g. an EEPROM read event */
   ...
  }
}

EEPROM_TASK(xyz)
{
  vuint8 lDTC[3];

  ...
  /* Wait for EEPROM event */
  /* EEPROM event is finished with reading */
  {
    DescRingBufferWrite(g_iContext, &lDTC, 3);
  /* Now trigger next EEPROM reading */
  }
}
```

7.6.2 with synchronous write




```
//2. Read Service (with synchronous Ring-Buffer)
// - static length
// - sub-function/PID

extern void ApplDescReadDTC_AddOn(DescMsgContext* pMsgContext);

void DESC_API_CALLBACK_TYPE ApplDescReadDTC(DescMsgContext* pMsgContext)
{
  vuint8 lData;
  /* Format check already done by CANdesc */
```

```
  /* Analysis of request has to done by ECU application */

  /* Set the response length */
  pMsgContext->resDataLen = 16;

  /* Fill the first data */
  lData = 5;

  /* check only on services with sub-function (e.g. 0x19) */
  if(pMsgContext->msgAddInfo.suppPosRes != 0)
  {
    /* since no response required – skip further processing */
    DescProcessingDone(pMsgContext->iContext);
  }
  else
  {
    /* Now we have to set CANdesc into the Ring-Buffer mode */
    DescRingBufferStart(pMsgContext->iContext);
    /* Now it is possible to write into the Ring-Buffer */
    DescRingBufferWrite(pMsgContext->iContext, &lData, 1);

    /* Use RepeatedSeriveCall feature to poll e.g. EEPROM driver */
    DescStartRepeatedServiceCall(pMsgContext->iContext, &ApplDescReadDTC_AddOn);
  }
}

void ApplDescReadDTC_AddOn(DescMsgContext* pMsgContext)
{
  vuint8 lDTC[3];
  DescMsgLen freeSpace;
  /* Check if enough space is free in ring-buffer */
  freeSpace = DescRingBufferGetFreeSpace();
  if (freeSpace >= 3)
  /* try to read from EEPROM */
  {
  /* Success - result is in lDTC */
    DescRingBufferWrite(pMsgContext->iContext, &lDTC, 3);
  }
  else
  {
    /* nothing to do, wait for next MainHandler call, ring-buffer is full */
  }
}
```

7.7 …prevent the ECU going to sleep while diagnostic is active

Most car manufactures have the requirement to keep the ECU alive while the diagnostic layer is active; including a pending request or a non-default session is currently active.

This requirement is handled by CANdesc for some car manufactures (see OEM specific TechnicalReference_CANdesc document for details)

The following code example shows all necessary steps to keep the ECU alive while diagnostic jobs are running (e.g. non-default session):

```
{
  DescContextActivity lActivity;
  DescStateGroup lState;
```

```
  lAcitvity = DescGetActivityState();
  lState = DescGetStateSession();

  /* check for a pending request or a non-default session */
  if ( ((lState & kDescStateSessionDefault) == 0) ||
       (lActivity != kDescContextIdle) )
  {
    /* Force to stay alive */
  }
  else
  {
    /* Ready for sleeping */
  }
}
```

7.8 …send a positive response without request after FBL flash job

According to the DC ECU programming specification after successful flashing of the ECU the application shall send a positive response either to “diagnostic session control – default session” or “ECU reset – hard reset” immediately after restart of the application. The Vector Flash Boot Loader will set a flag (reset response flag) in RAM or EEPROM which has to be evaluated by the application at startup. Depending on its value the application has to call the CANdesc function DescSendPosRespFBL with the appropriate response ID.

CANdesc provides the API `DescSendPosRespFBL` for this purpose.

Due to bus communication is necessary to send the positive response; some limitations have to be handled by the application:

1) Bus communication is to be requested by the application

2) If bus communication is possible, the application has to call `DescSendPosRespFBL`. CANdescBasic will send the positive response.

3) The application will be called to provide the concrete addressing information of the response.

4) Bus communication can be released by the application.

7.9 …enforce CANdesc to use ANSI C instead of hardware optimized bit type

CANdesc uses per default the bit-type definition provided by the CANdriver, since it is selected as optimal for the concrete CPU. On this way the CANdesc ROM and RAM resource consumption is kept as low as possible.

Due to the complexity of some CANdesc data structures there can be problems on certain compilers with special bit-structure compiler options.

If you encounter such problems either at compile or at run-time, you can turn the ANSIC C bit-type support in CANdesc on. To do that, just add a user configuration file in GENy with the following content:

```
#define DESC_USE_ANSI_C_BIT_TYPE
```

8 Related documents

Abbreviation	File Name	Description
/KWP2000/		Keyword 2000 protocol
/TPMC/		User manual of the multi-connection transport layer module. The transport layer is implemented according to /ISO 15765/
/ISO 15765/		This ISO standard describes diagnostics and diagnostics on CAN.

Note: If no file name is given, the document is not provided by Vector.

9 Glossary

Abbreviation	Description
CANdb	**CAN d**ata**b**ase by Vector which is used by Vector tools.
CANdesc	**CAN d**iagnostics **e**mbedded **s**oftware **c**omponent
CDD	**C**ANdela **D**iagnostic **D**atabase
CF	**C**onsecutive **F**rame (transport protocol frame)
CCL	**C**ommunication **C**ontrol **L**ayer
DBC	CAN database format of the Vector company, which is used by the GENtool to gather information about the ECUs in the network, their communication relations, message definitions, signals of messages, network related information (e.g. manufacturer type, network management type, etc.).
ECU	**E**lectronic **C**ontrol **U**nit
FBL	**F**lash **B**oot **L**oader
KWP 2000	**K**ey**w**ord **P**rotocol **2000**
OSEK	German abbreviation, "**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug", means "open systems and the corresponding interfaces for automotive electronics"
RCR-RP	**R**equest **C**orrectly **R**eceived **– R**esponse **P**ending
SF	**S**ingle **F**rame
SID	**S**ervice **Id**entifier
SPRMIB	**S**uppress **P**ositive **R**esponse **M**essage **I**ndication **B**it
TP	**T**ransport **P**rotocol
UDS	**U**nified **D**iagnostic **S**ervices

10 Contact

Visit our website for more information on

- News
- Products
- Demo software
- Support
- Training data
- Addresses

www.vector-informatik.com