CANdesc

Technical Reference

Version 3.07.00

Authors	Oliver Garnatz, Mishel Shishmanyan, Stefan Hübner, Matthias Heil, Katrin Thurow, Patrick Rieder
Status	Released

1 History

Author	

			- 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:** -12.6.13 DynamicallyDefineDataIdentifier ($2C) (UDS) functions -12.6.13.1 DescMayCallStateTaskAgain()
Mishel Shishmanyan	2007-02-22	2.12.00	**Added:** *- 12.6.9.3 "DescRingBufferCancel()"*
Matthias Heil	2008-01-03	2.13.00	**Added:** Caution concerning user main handler on protocol level I
Matthias Heil	2008-02-29	2.14.00	**Added:** Handling of read/write memory by address: *- 9.3 "Read/Write Memory by Address"* *- 12.6.8.2 "DescStartMemByAddrRepeatedCall()"* *- 12.6.14 "Memory Access Callbacks"*
Mishel Shishmanyan	2008-06-06	2.15.00	***Removed:*** *Chapter "ResponseOnEvent Transmission Unit"* ***Added:*** *- 12.6.13.3 "Non-volatile memory support"*
Mishel Shishmanyan	2008-11-09	2.16.00	***Modified:***

			- *12.6.9* and *12.6.9.1*: Added limitation for UDS and SPRMIB with the ring buffer usage. - *13.6 …work with the ring-buffer mechanism* ***Added:*** - *12.6.15 Flash Boot Loader Support* - *13.8 …send a positive response without request after FBL flash job*
Mishel Shishmanyan	2009-05-18	2.17.00	***Modified:*** *12.6.6.1ApplDescCheckSessionTransition()* ***Added:*** *12.6.6.3DescIsSuppressPosResBitSet ()*
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:*** *13.9 …enforce CANdesc to use ANSI C instead of hardware optimized bit type* *5.1 Configure DBC attributes for diagnostics*
Mishel Shishmanyan	2009-09-17	3.00.00	***Added:*** *6 CANdesc Configuration in GENy* *8 Multi Identity* *12.6.2 Multi Variant Configuration Functions*
Mishel Shishmanyan	2010-01-26	3.01.00	***Added:*** *7 CANdescBasic Configuration in GENy*
Mishel Shishmanyan	2010-12-21	3.02.00	***Modified:*** *12.6.14.1 ApplDescReadMemoryByAddress()* *12.6.14.2 ApplDescWriteMemoryByAddress()* *12.6.9.2 DescRingBufferWrite()*
Katrin Thurow	2011-08-25	3.03.00	***Added:*** *8.1* Single Identity Mode *8.3* Multi Identity Mode *13.10* …configure Extended Addressing

			13.11 …use Multiple Addressing *12.6.6.7* DescGetSessionIdOfSessionState ***Modified:*** *8* Multi Identity Support *13.8* …send a positive response without request after FBL flash job
Katrin Thurow	2011-09-19	3.04.00	***Added:*** *13.12*…use “Dynamic Normal Addressing Multi TP” with multiple tester ***Modified:*** *13.11* …use Multiple Addressing
Katrin Thurow	2011-11-27	3.05.00	***Added:*** *12.6.17* “Spontaneous Response” transmission ***Modified:*** *6.2.1* Global CANdesc Settings
Patrick Rieder	2013-01-23	3.06.00	***Added:*** *10* Generic Processing Notifications *12.6.18* Generic Processing Notifications ***Modified:*** *6.2.1* Global CANdesc Settings *12.6.4* Service callback functions *12.6.9* Ring Buffer Mechanism Small fixes
Patrick Rieder	2013-05-27	3.07.00	***Added:*** *11* Busy Repeat Responder Support ***Modified:*** *13.12* …use “Dynamic Normal Addressing Multi TP” with multiple tester

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

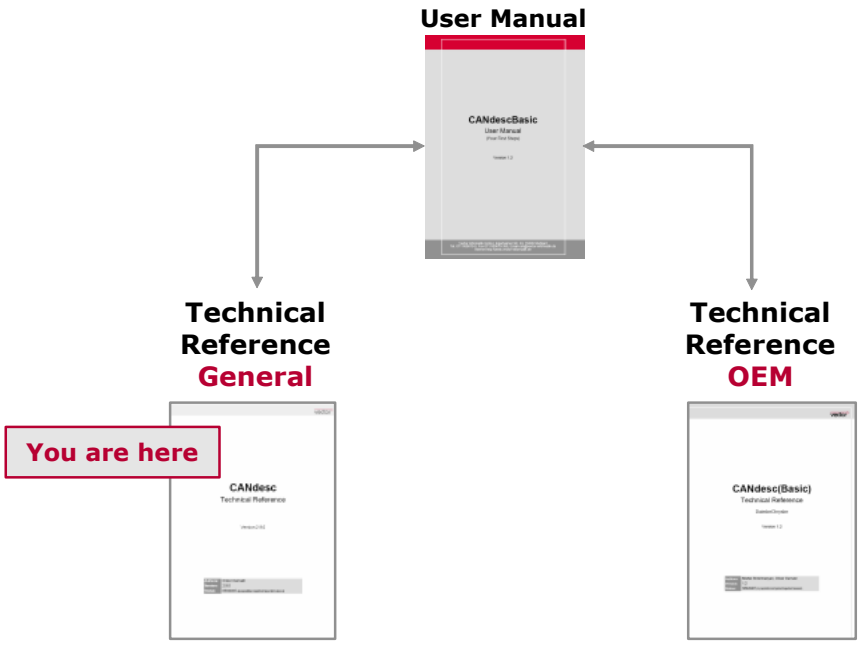


Figure 3-1: Manuals and References for CANdesc

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.

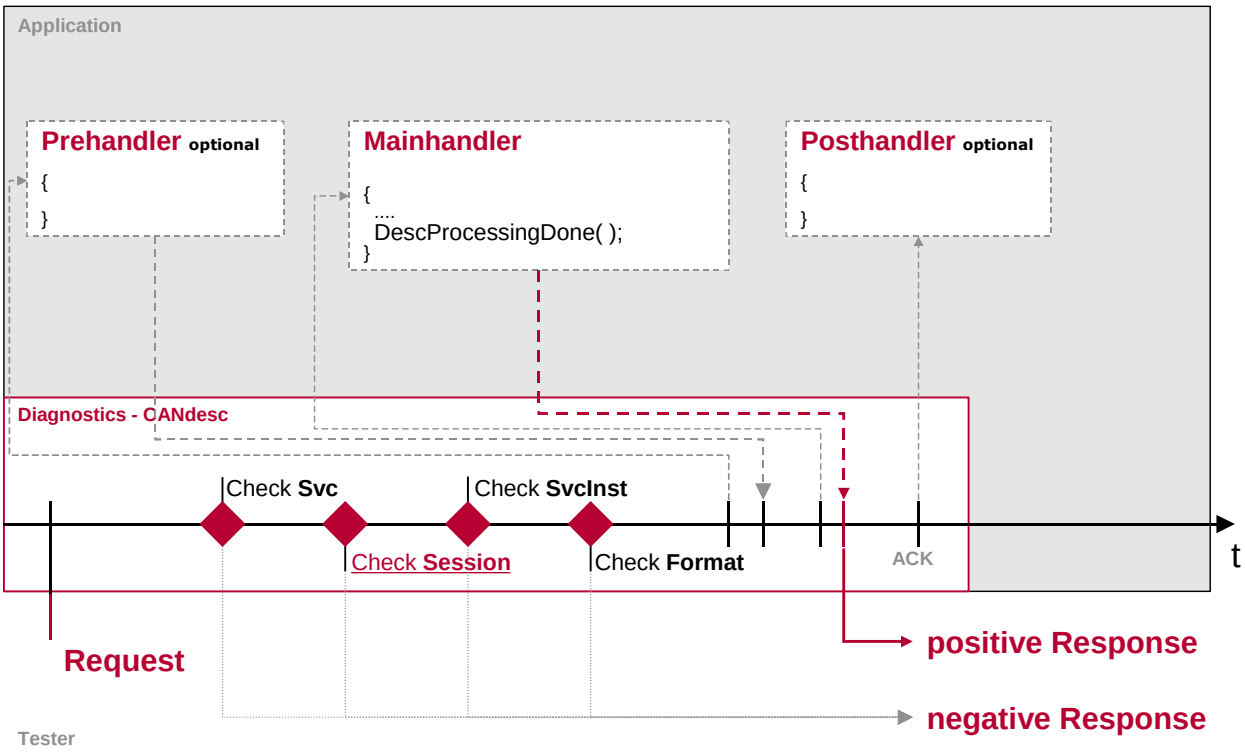


Figure 4-1: General request flow

4.1.2 How does this flow actually work?

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

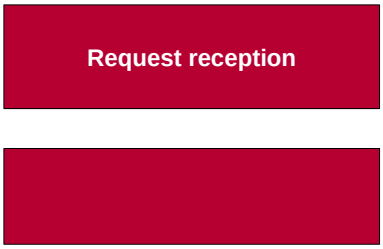


Figure 4-2: DESC run diagram

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.

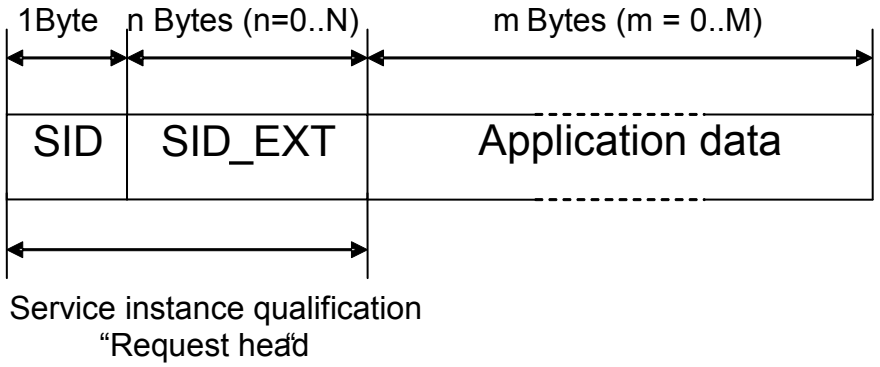


Figure 4-3: Request message mapping

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 the 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.

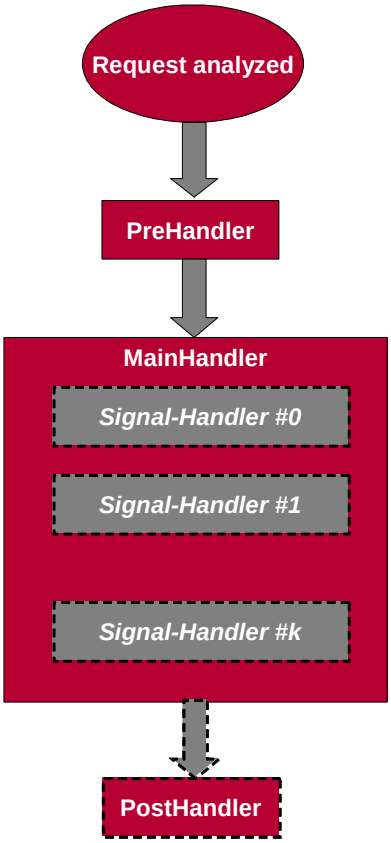


Figure 4-4: Request processing stages

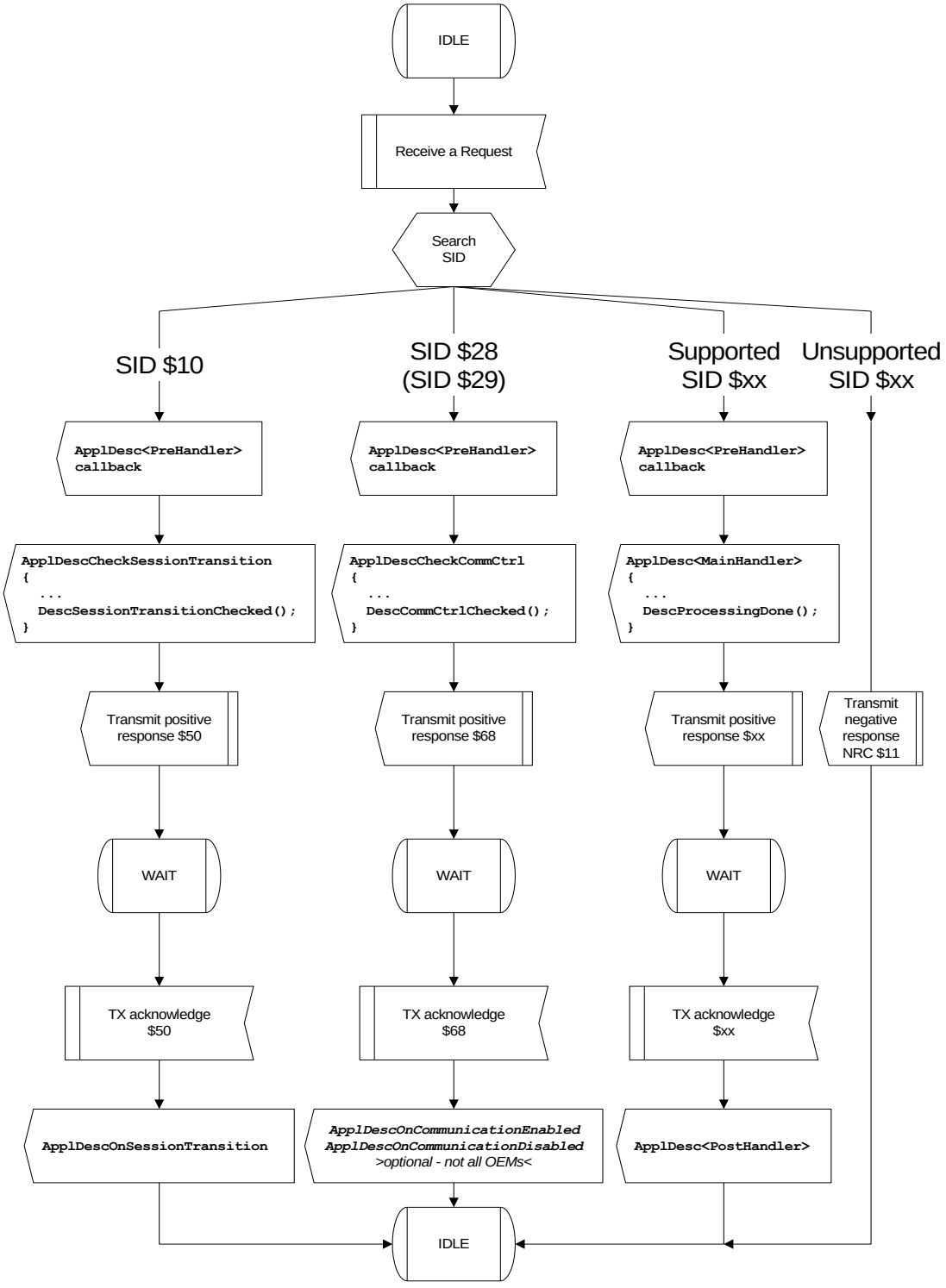
After the response is composed CANdesc must be informed about, to start the transmission of the final response. CANdesc is doing the handshake with the Tester (automatic transmission of RCR-RP) while the state ***"Processing the request"*** is active.

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 12.6.6 Session Handling and in chapter 12.6.7 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

6 CANdesc Configuration in GENy

Since version 6.00.00, the CANdesc configuration concept has been improved by splitting the concrete ECU parameterization and software integration from the diagnostic specification.

The configuration of CANdesc in GENy consists of two important steps:

- Importing a diagnostic description file. Currently only CANdela (CDD) files are supported therefore in further only the term CDD file will be used.
- Setup all service options required by the application like:
  - Configure the service handlers (pre-, main- and post-handlers)
  - Setup the service specific settings, like maximum number of dynamically defined items per DynDID, size of scheduler for periodic data reading, etc.
  - Setup timing parameters (e.g. periodic rates).

The second step is optional, since after importing a CDD file all important settings will be already prepared for usage. If there are missing or invalid settings, GENy will notify you at generation time.

6.1 Step One – Importing an ECU Diagnostic Description

After activating the CANdesc component in GENy, you will have the following view:

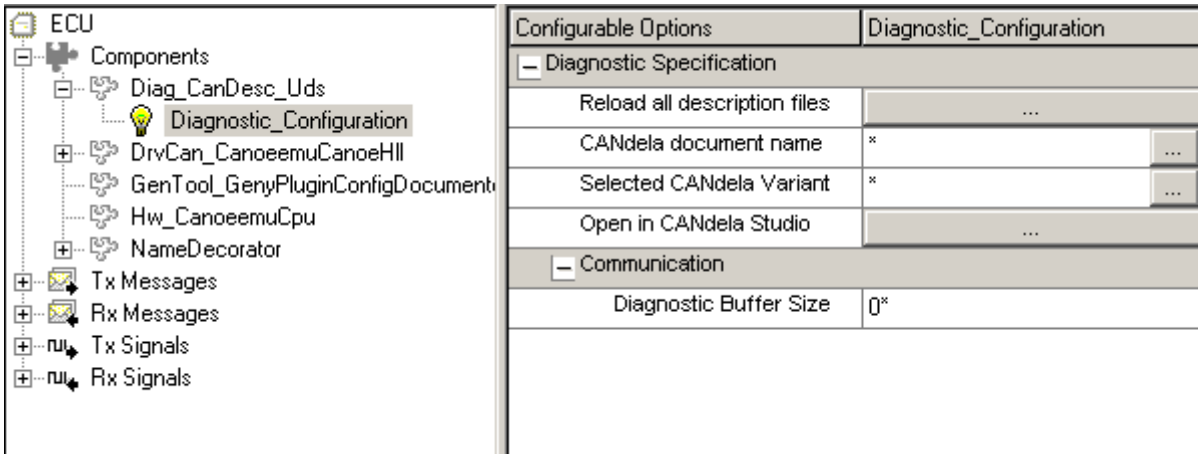
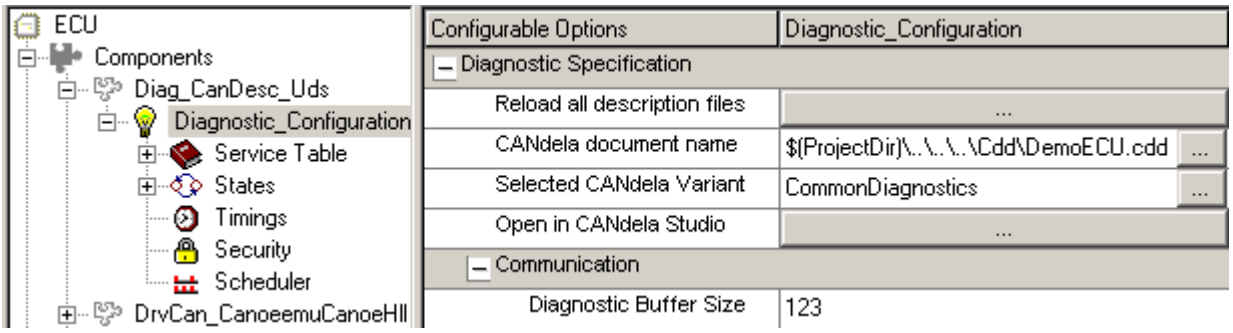


Figure 6-1 CANdesc GENy startup screen

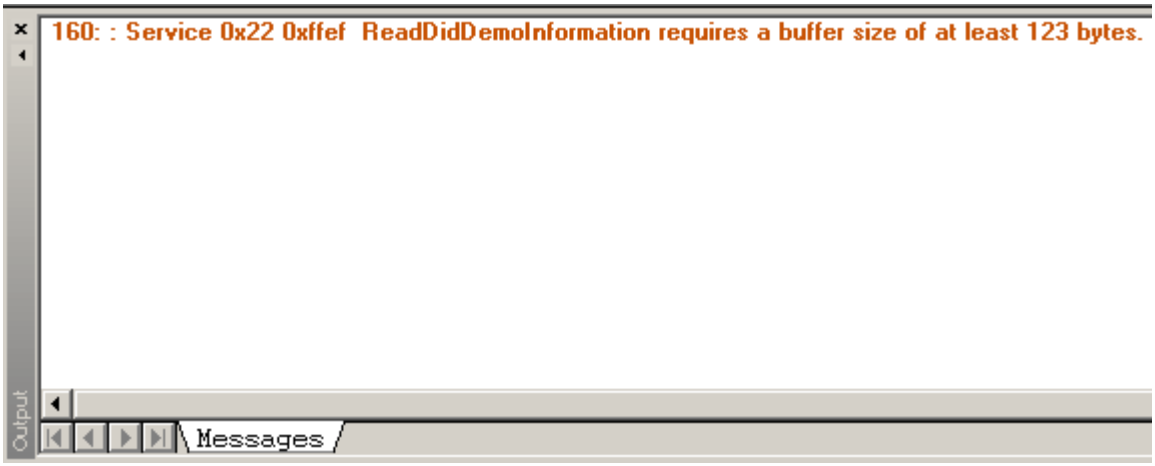
At this time GENy does not have any CDD file and can not generate CANdesc. You have to specify a CDD file, using the button on the option “CANdela document name”.

After selecting the CDD file, the CANdesc component tree view will look like:



Info

Please note, the diagnostic buffer size is now set to a non-zero value. At CDD import time, GENy calculates a statistic over all services with simple, linear data structure and sets the buffer size to fit the longest request resp. response message. The message window will show you which service requires the suggested buffer size:



Complex services like reading the faultmemory information or upload/download/transferdata are excluded from this statistic, since the worst case response calculation is not possible.

You can still set another value for the buffer size, even lower as the size suggested by GENy. At generation time, the code generator will check again the set buffer size and consider more options you have changed (like RingBuffer support) and notify you if the buffer size is too small.

Now you can try to generate your diagnostic layer, using the default settings.

6.2 Step Two – ECU Diagnostic Configuration in GENy

Once the CDD content is imported, there are several options that can and shall be set up for best match on your ECU integration needs.

What You Can Configure in GENy

The goal of splitting the ECU integration configuration from the ECU diagnostic specification is to provide a simplified view on what the ECU diagnostic application developer is able to configure without danger of changing the diagnostic specification provided by the OEM.

If a CANdesc parameter is not available in the source diagnostic description (CDD file), you will be able to edit it in GENy, even if it is relevant for the diagnostic specification.

The chapters below will show you all configuration parameters of CANdesc that can be set up in GENy.

What You Can Not Configure in GENy

All diagnostic parameters that could affect the ECU behaviour regarding its diagnostic specification, provided by the concrete OEM or would lead to inconsistency between the tester expectations on the ECU behaviour are not editable in GENy. If a change is required on such a parameter, the diagnostic description source shall be modified, to guarantee that the OEM or/and the tester will take this change into account.

6.2.1 Global CANdesc Settings

Under the generic settings you will find the options that affect the overall module performance, independently of the diagnostic services that shall be supported. In the picture and the table below follows the description of the settings for CANdesc.

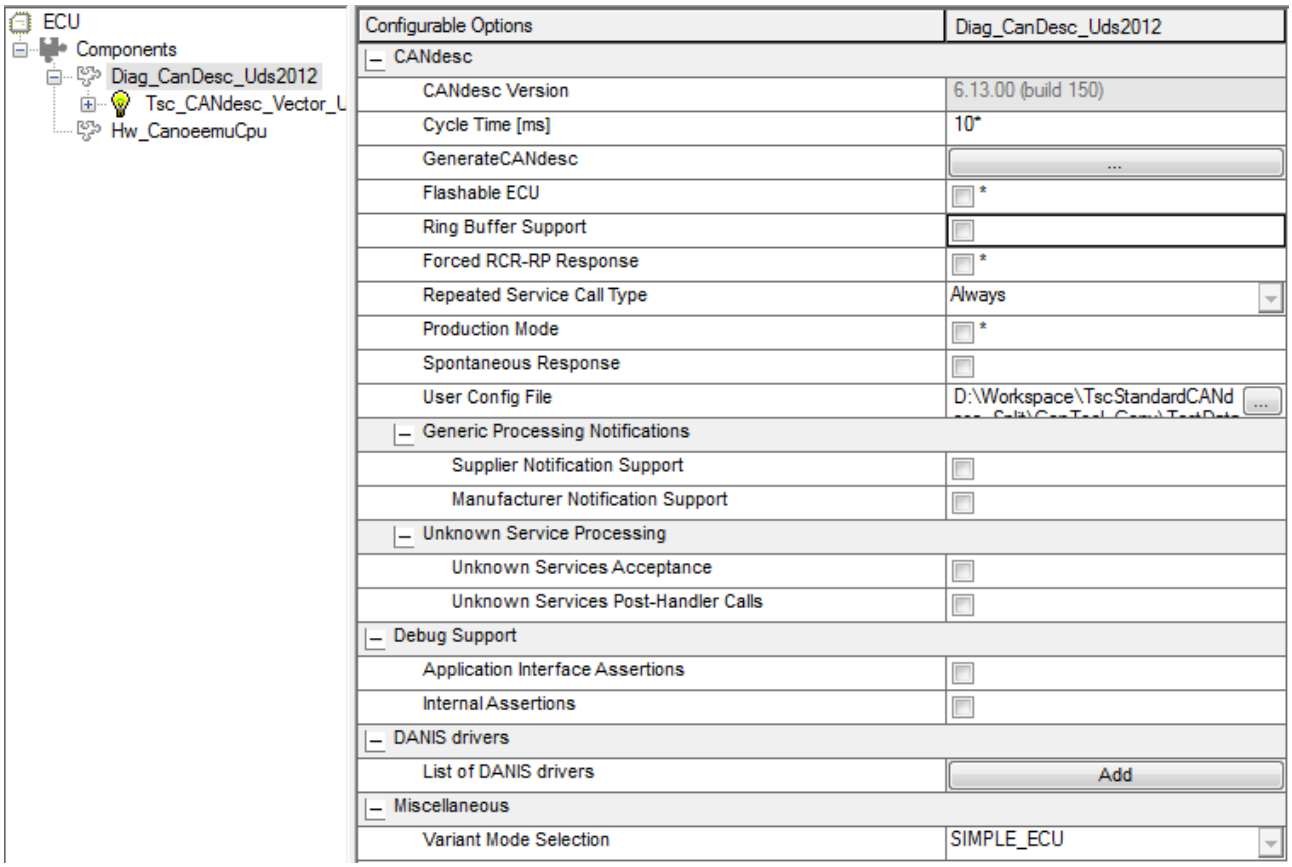


Figure 6-2 Example of GENy global CANdesc settings

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
Cycle Time [ms]	Always available.	Integer	**10** 1..255	The DescTask (resp. DescTimerTask) function must be called EXACTLY in the time period specified here. This is important since the time constant will be converted into a number of function calls and if this setting doesn't match the real call cycle, the component internal timeout monitors will not function properly.
Generate CANdesc	Always available.	Button		This feature is only available after you have generated the whole CANbedded package. NOTE: If you run into problems, generate the whole package again!
Number of 'Busy-RepeatRequest' responded Requests	OEM dependent availability.	Integer	**0** 0..255	The value is the maximum count of parallel handled diagnostic requests. Only the first diagnostic request will be processed, all other (additonal) request, which will be received while the first one is in process, will be also received, but only responded with NRC $21 ('BUSY - repeat request'). If there are more requests onto the bus than this number, only the first N will be responded - all other will be just ignored.
Flashable ECU	OEM dependent availability.	Boolean	**False** True	Depending on the car manufacturer this option has different effects. Please, see the OEM specific technical reference document for more information.
Ring Buffer Support	Always available.	Boolean	**False** True	In case your ECU shall send a very long positive response for some services (usually when reading fault memory) you can reserve enough RAM for the diagnostic buffer to handle the longest possible response length, or you can use the built-in ring-buffer mechanism which allows usage of smaller buffer. The linear buffer usage saves ROM and run-time but needs more RAM, the ring-buffer saves RAM (you may send 4095 Byte response with a 20Byte buffer) but requires more ROM and causes run-time overhead when used. NOTE: This

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
				option just unlocks the built-in support, but the selection usage of the feature is done at run-time by your application (for each service independently).
Forced RCR-RP Response	OEM dependent availability.	Boolean	**False** True	In some cases (e.g. prior jump into the FBL (FlashBootLoader), ECU busy so no task function can be called for long period of time) it is necessary to prevent the tester from ECU response timeout. Enabling this feature you will be able to send a RCR-RP (ResponseCorrectlyReceived-ResponsePending) response any time during an active serivce processing (main-handler called but no DescProcessingDone has been called yet).
Repeated Service Call Type	Always available.	Enum	**Deactivated** Always Individual	In some cases (usually for slow services like reading from EEPROM) it is useful to let the component to poll your application (service main-handler) until the service execution is completed. Otherwise you have to leave the service's main-handler function and trigger an own additional polling task and finalize the service from there. Using the built-in polling mechanism you will save ROM and run-time. Also it prevents from confusing code structures. Always: Each main-handler will be called as long as the application didn’t call *DescProcessingDone*(). Individual: Each main-handler will decide by itself if it will be called once or as long as the application didn’t call *DescProcessingDone*().
Production Mode	OEM dependent availability.	Boolean	**False** True	Enabling the production mode will set all options in the possible safest (uncritical) value. Some car manufacturers don't allow all of the features in production, so they will be turned off.
Spontaneous Response	Available if Service 0x86 is part of the diagnostic configuration.	Boolean	**False** True	This setting enables the possibility to send diagnostic responses without a preceding request. This feature is needed for Service 0x86 with Transmission Type I.

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
				The spontaneous response can be triggered via the API DescSendApplSpontaneousResponse.
Supplier Notification Support	Available if CANdesc according to ISO 14229-1 2012 is used.	Boolean	**False** True	If this option is enabled, CANdesc notifies the application on incoming service requests and outgoing responses. CANdesc only notifies the application if the requested service is supported in the active session and security state. For more details see *10 Generic Processing Notifications*
Manufacturer Notification Support	Available if CANdesc according to ISO 14229-1 2012 is used.	Boolean	**False** True	If this option is enabled, CANdesc notifies the application on incoming service requests and outgoing responses. CANdesc notifies the application right before the processing of the request starts and after a response has been sent. For more details see *10 Generic Processing Notifications*
Unknown Services Acceptance	OEM dependent availability.	Boolean	**False** True	In some cases if the diagnostic database doesn't contain all necessary service Ids, or you need a (some) test identifier(s), you can enable this option which will redirect all received requests with unknown service Ids to your application for additional acknowledgment and processing.
Unknown Services Post Handler Calls	OEM dependent availability.	Boolean	**False** True	If the option 'Unknown Services Acceptance' is enabled, you may use this feature to be notified each time an unknown service processing has been accomplished. This post handler usage is the same as the one of the normal services post handlers.
Application Interface Assertions	Always available.	Boolean	**False** True	The SW component provides built-in debug support (assertion) to ease up the integration and test into the project. In general, the usage of assertions is recommended during the integration and pre-test phases. It is not recommended to enable the assertions in production code due to increased runtime and ROM needs. The assertion checks the correctness of the assigned

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
				condition and calls an error-handler in case this fails. The error handler is called with an error and line number. You can find information about the defined error numbers in the Desc.h file.
Internal Assertions	Always available.	Boolean	**False** True	The SW component provides built-in debug support (assertion) to ease up the integration and test into the project. In general, the usage of assertions is recommended during the integration and pre-test phases. It is not recommended to enable the assertions in production code due to increased runtime and ROM needs. The assertion checks the correctness of the assigned condition and calls an error-handler in case this fails. The error handler is called with an error and line number. You can find information about the defined error numbers in the Desc.h file.
List of DANIS drivers	Always available.	String List		Add an arbitrary list of DANIS drivers for custom bus access. Each entry here will result in a user driver, which can be used to connect CANdesc to arbitrary transport layers. Example: Adding a driver name “MostTp” will force CANdesc to generate templates for a driver with this name. You will have only to implement the functions of the driver skeleton.
UUDT Message Confirmation Timeout [ms]	Available only if UUDT message transmission is supported.	Integer	**100** 1..65535	This is the maximum time after which a UUDT (Unacknowledged Unsegmented Data Transfer) message will be deleted from the CAN drive request queue and (if possible) will be replaced by the next queued message.
Faultmemory Iteration Limiter	Available only if CANdesc provides fault-memory service	Integer	**0** 0..255	Limit the iteration depth for faultmemory read services.

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
	implementation.			Some faultmemory ($19) services can consume much runtime when performed en bloc. To reduce the run time of the CANdesc task, use this option to limit the iteration depth of the faultmemory access function so your controller can handle the workload. ATTENTION: Depending on your Tp timeout settings, to low a number of iterations can result in an aborted transmission due to buffer underrun. A value of 0 (zero) will disable any limitation.
Variant Mode Selection	OEM dependent availability.	Enum	**None** Multi Identity Mode VSG Mode	Note: This setting is independent from communication identities! None: The diagnostics support one configuration only. Multi Identity Mode: The diagnostics support different diagnostic variants. One variant is active a time. VSG Mode: Diagnostic Entities (SubServices, DTCs...) are grouped into VSGs. Several VSGs can be active at a time.

6.2.1.1 Generic Processing Notifications (UDS2012)

On activation of the feature “Generic Processing Notifications”, GENy shows the names of the additional callbacks that will be generated. The names of the callbacks are fixed and can not be modified (see Figure 6-3). For a detailed description of the feature see chapter 10 Generic Processing Notifications.

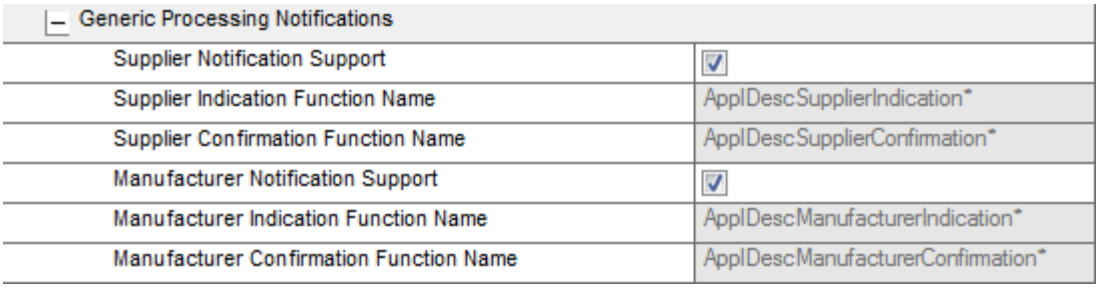


Figure 6-3 Activated feature “Generic Processing Notifications”

6.2.2 Service Specific Settings

Once the CDD file is imported you can have an overview of the supported services of your ECU:

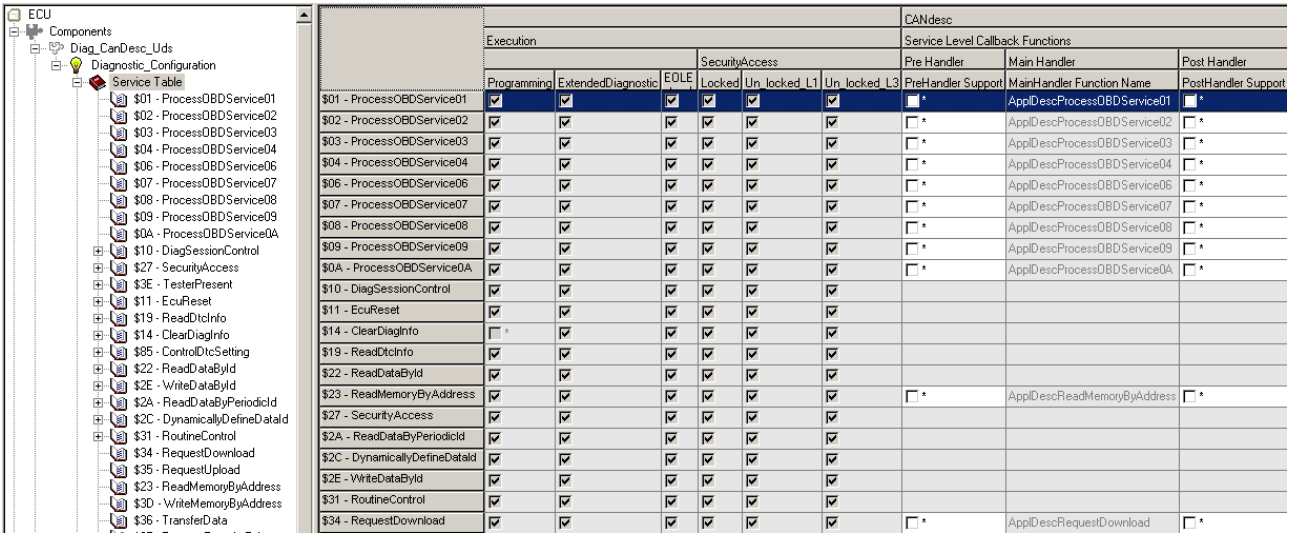


Figure 6-4 GENy diagnostic service overview

On this level you can also configure all services that will be supported on service Id level only.

6.2.2.1 Generic Service Settings

Using the CANdesc component tree view you can explore the detailed settings for each service and its sub-services (if available).

A generic sub-service setup looks like the picture below:

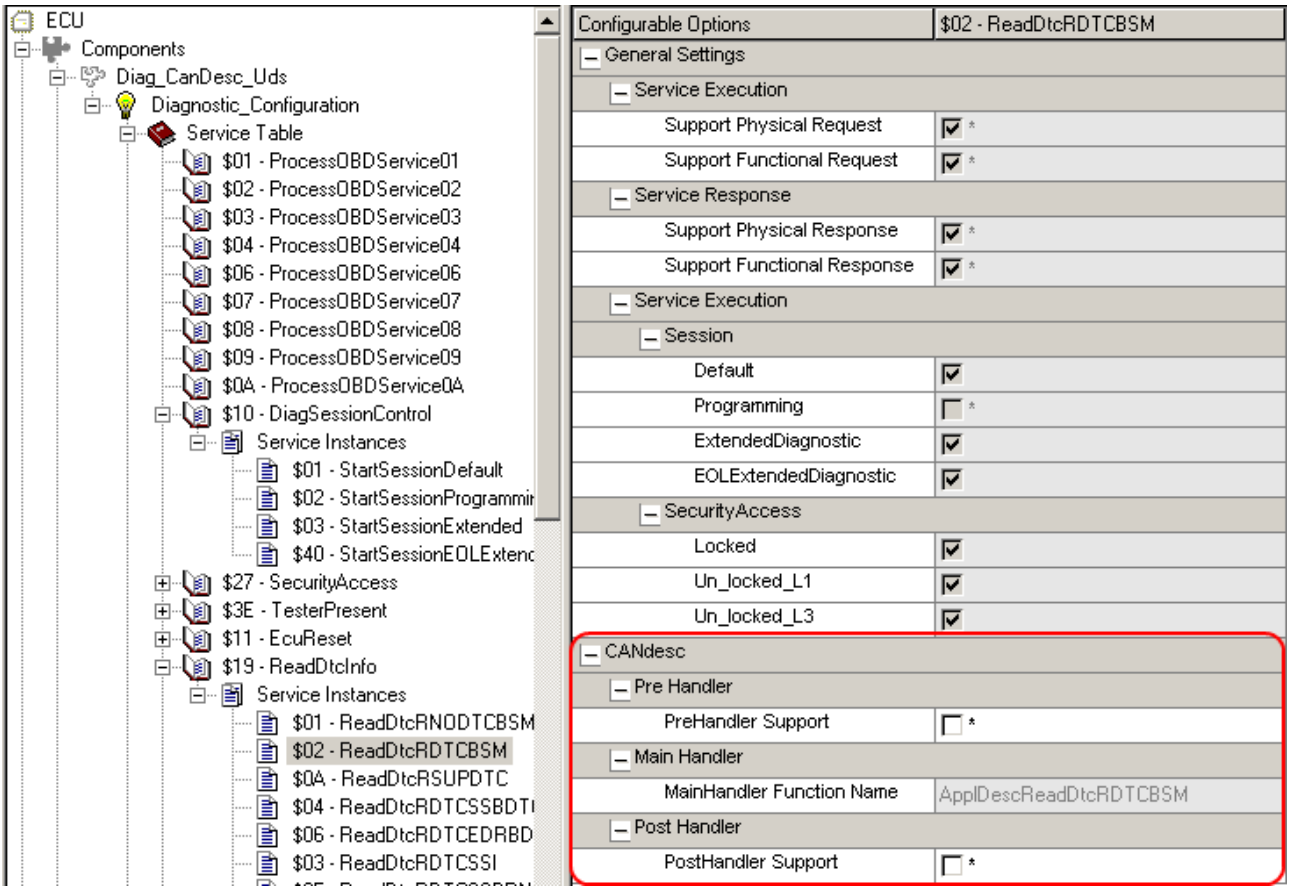


Figure 6-5 GENy generic sub-service setup

Almost all services have a very simple configuration view. You can see the main-handler is always available and a preview of the call-back name is shown.

You can only add a pre- and / or a post-handler to such a service, if required.

6.2.2.2 Predefined (implemented) Services in CANdesc

There are configurations (OEM dependent) where several services are fully implemented by CANdesc. Such service can be, StartDiagnsoticSession, SecurityAccess, DynamicallyDefinedDataIdentifier, ReadDataByPeriodicIdentifier, etc.

Those services that will not be handled by the application are marked in GENy as shown on the picture:

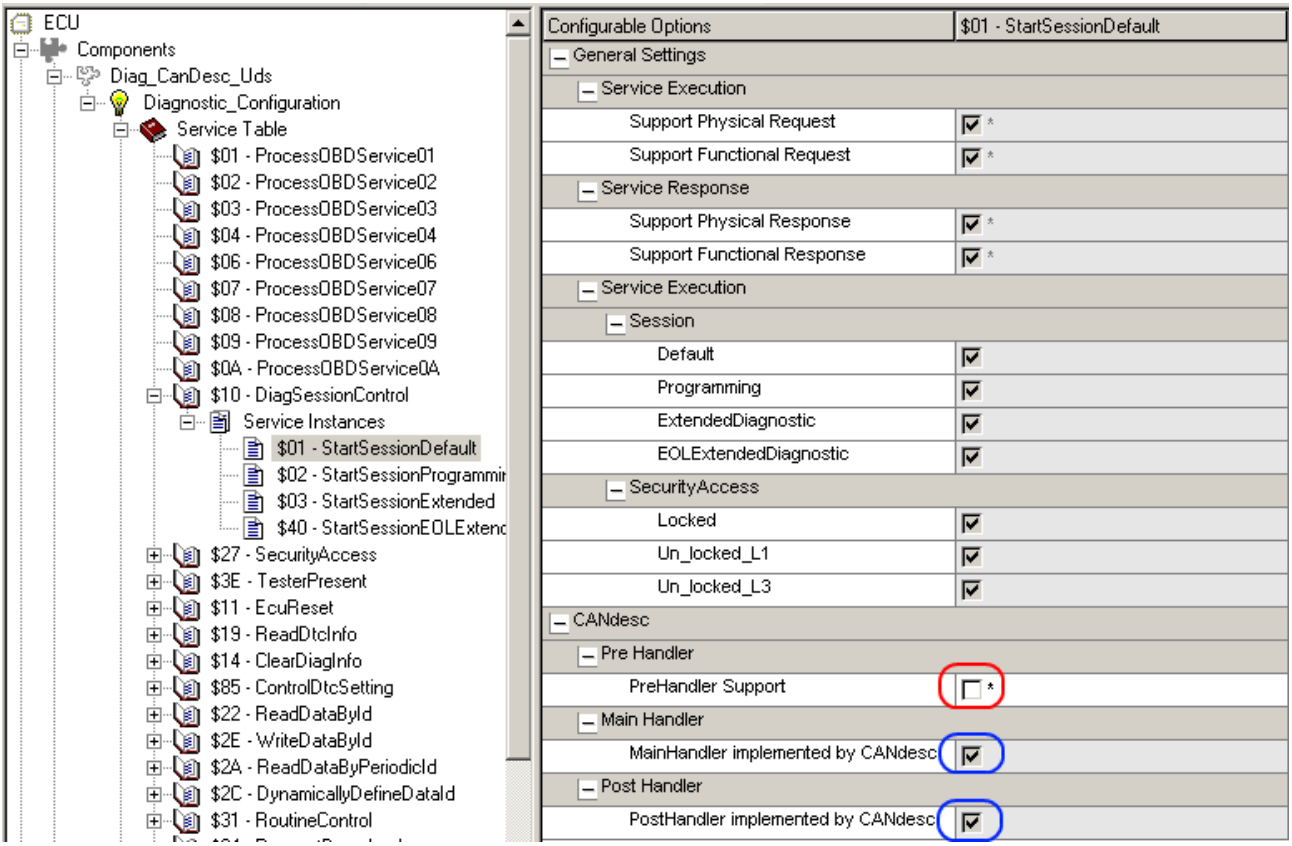


Figure 6-6 GENy predefined sub-service setup

As you can see, the main-handler is grayed and marked as “implemented by CANdesc”. The same can apply (depends on the service) also to the pre- and post-handlers of the service.

In the example on the *Figure 6-6 GENy predefined sub-service setup* you see that the pre-handler is still free for usage. This means you can still implement a pre-handler to check additional conditions prior CANdesc will be able to process the service. For other service it could be also the post-handler free for implementation.

There are several services that make some exceptions to the predefined implementation rule:

Service 0x2A:

- PreHandler configuration is possible: If a pre-handler is required, it must be enabled on all sub-functions of the concrete DID. The pre-handler name will be “ApplDescPreReadPeriodicDid<DID instance name>”.
- PreHandler on “stop all” is not used by CANdesc and will not be considered during the code generation even if it is enabled.
- Main-Handler are set to “implemented by CANdesc” since the data reading call-back will be the corresponding 0x22 DID service call. This means that if the corresponding service 0x22 DID has been set to use the “Signal API”, the periodic reading service will use it too.
- Post-Handlers are not supported at all.

Service 0x2C:

- PreHandler configuration is possible: If a pre-handler is required, it must be enabled on all sub-functions of the concrete DID. The pre-handler name will be “ApplDescPreDynDefineDid<DID instance name>”.
- PreHandler on “clear all” is not used by CANdesc and will not be considered during the code generation even if it is enabled.
- Main-Handler are set to “implemented by CANdesc” since the DID definition is always done by CANdesc.
- Post-Handler are not supported at all.

6.2.2.3 Signal Access Enabled Services

Some services such as the UDS ones 0x22/0x2A and 0x2E, can be processed on signal level. This means CANdesc will analyze the request/response data structure and generate the service main-handler, leaving to the application only the task to provide the signal values for the response, resp. to write the requested signal values to the ECU memory.

The setting view of such a service is shown below:

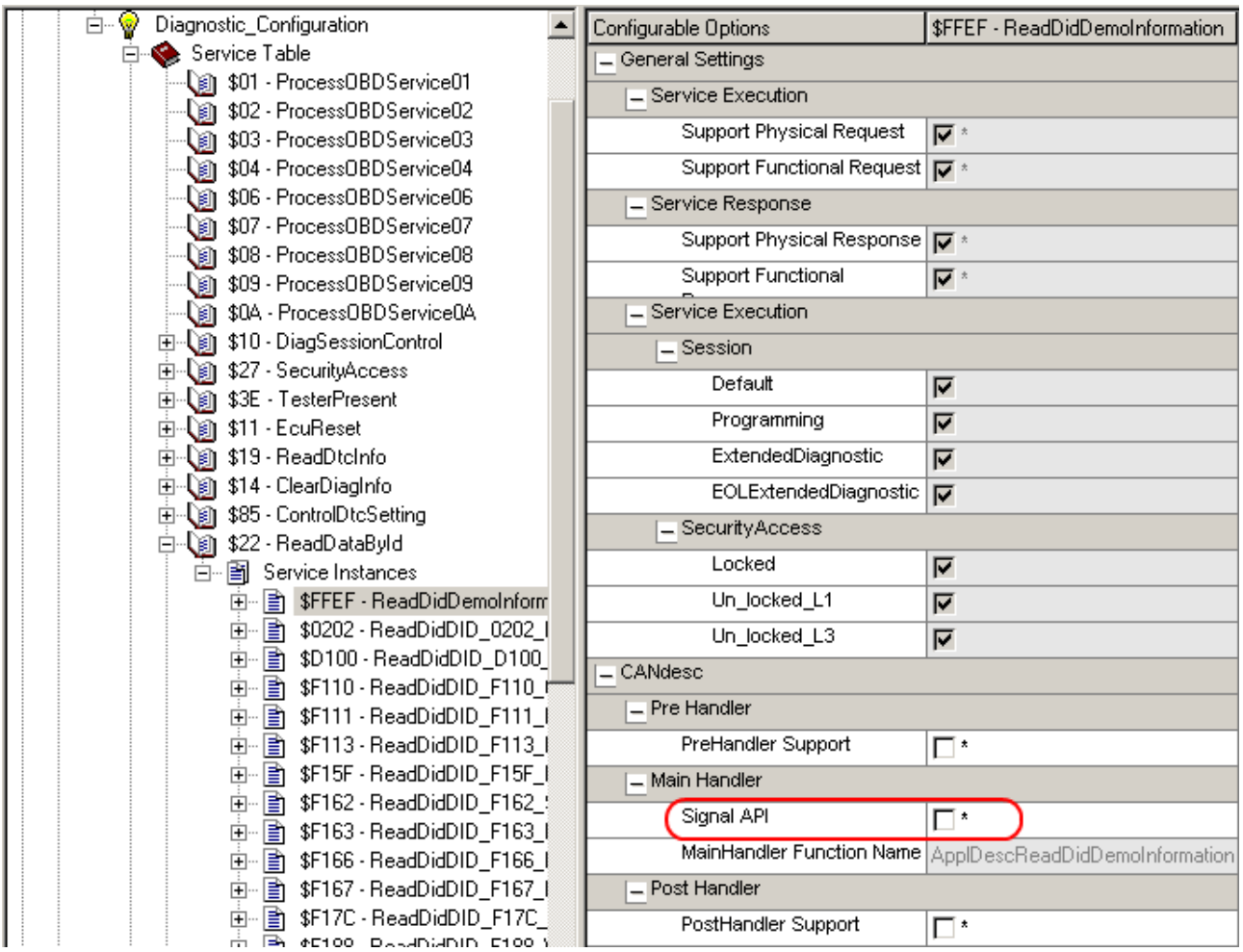


Figure 6-7 GENy signal API enabled sub-service setup

Note: For the read dynamically defined DID service, there is no signal access since they are always implemented by CANdesc internally.

If the “Signal API” option is not enabled this service is to be implemented like any other diagnostic service. The data object specific settings, described below, will have no effect on the code generation.

If the “Signal API” option is enabled, CANdesc will generate per default a call-back function for any data object (signal) the service contains. You can specify more options on each signal, to achieve the maximum advantage of CANdesc – fully implemented diagnostic service.

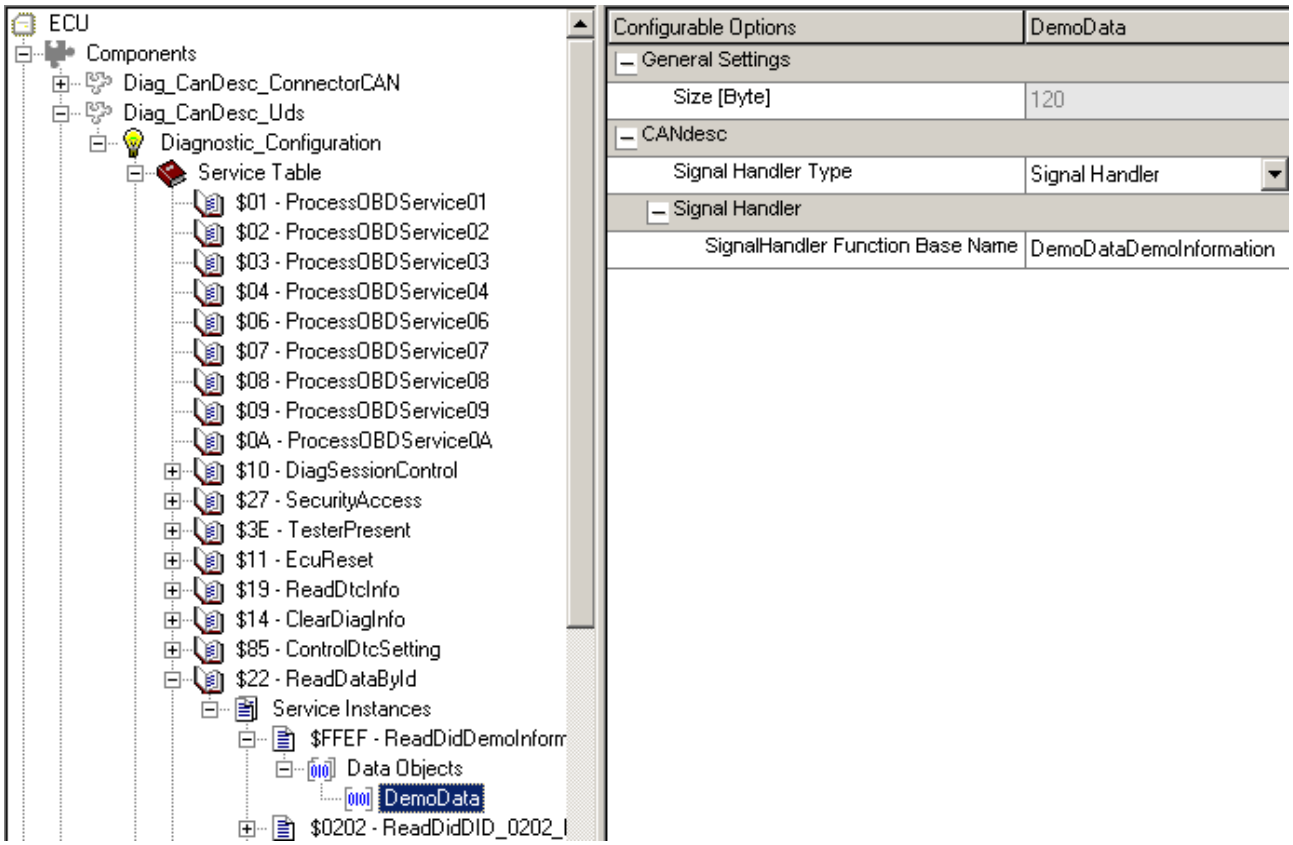


Figure 6-8 GENy signal view of a sub-service

You can have three types of signal handling:

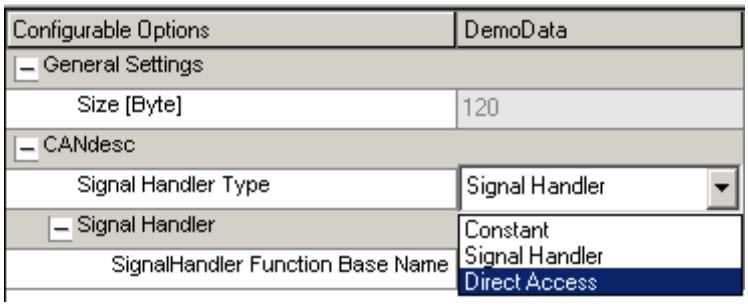


Figure 6-9 GENy signal handler types

FAQ

Constant is only possible if the CDD file has contained constant value for the selected data object. You can not specify in GENy a constant value for a signal handler.

In case of selected “Direct Access” signal handling, the following options will be enabled:

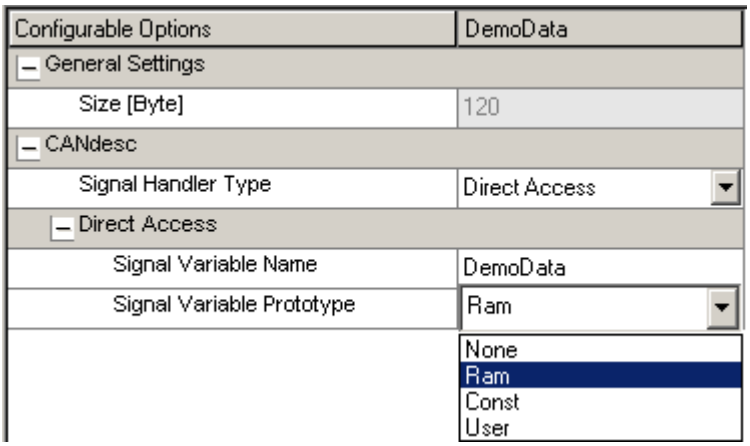


Figure 6-10 GENy direct access signal handler settings

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
Signal Handler Type	Only for signal API enabled services.	Enum	**SignalHandler** Constant DirectAccess	Select the type of signal handler Constant: The data value is constant. The data value can be used directly. This is used only when the corresponding subservice uses a signal API main handler. Signal Handler: Use a callback function to get/set the data value. This function is used only when the corresponding subservice uses a signal API main handler. Direct Access: Directly use a variable to access the data object. Also, a signal API main handler has to be used for this setting to have any effect.
SignalHandler Function Base Name	Only for signal API enabled services and a signal access through a **SignalHandler** is selected	String	**<DataObjectQualifier>+<DiagInstanceQualifier>**	This value is used as base for the signal access function - depending on how the value is used, the name entered here is prefixed with different prefixes, e.g ApplDescRead / ApplDescWrite. You can override the default name, by specifying an own signal base. The Prefix (e.g. ApplDesc can not be overridden).
Signal Variable Name	Only for signal API enabled services	String	**<DataObjectQualifier>**	The name of the signal variable.

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
	and if **DirectAccess** signal handling is selected			Example: c_dataTemp g_applData.bit0
Signal Variable Prototype	Only for signal API enabled services and if **DirectAccess** signal handling is selected	Enum	**Ram** None Const User	To create the proper extern declaration to access the signal variable, the proper access modifiers have to be specified. None: No prototype is generated at all. "DescType.h" where the user has to define the real typedefs (for structure access for example). Ram: The variable is located in RAM. Const: The variable is located in ROM. User: Set a user defined prototype.
Signal Variable User Prototype	Only for signal API enabled services and if **DirectAccess** signal handling is selected and if the Signal Variable Prototype is set to **User**	String	**Empty**	Set the prototype of the signal variable. Example: boolean EcuTempType

6.2.3 Timing Settings

GENy imports all possible timings that the diagnostic description source provides. Those parameters that are available in the CDD file are considered as a part of the ECU specification and are not modifiable in GENy. If a modification of those parameters is required, please change their values in the diagnostic description file and re-import it in GENy.

All other parameters can be set up manually, but the default value already matches the OEM specification.

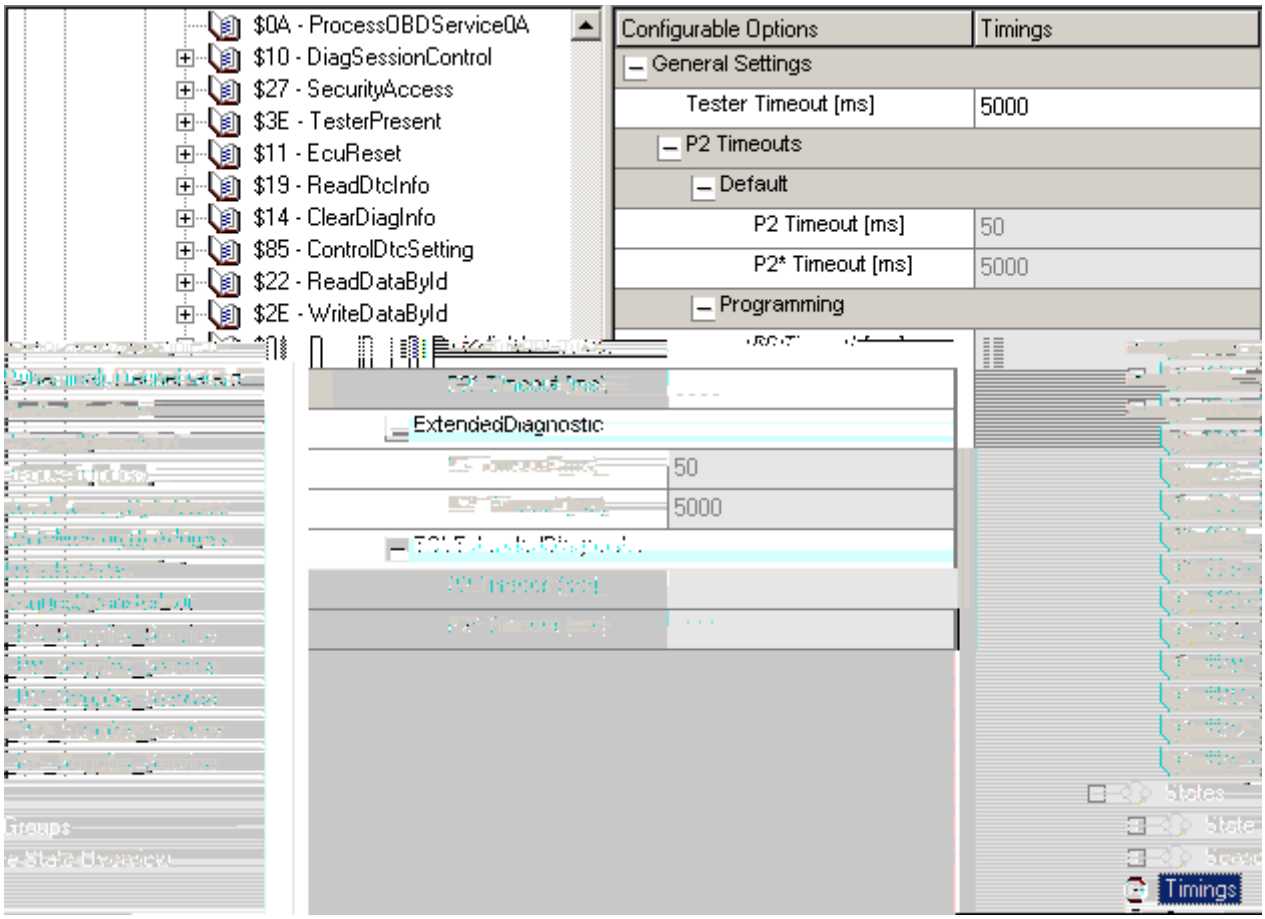


Figure 6-11 GENy CANdesc timing parameters

6.2.4 Security Access Settings (UDS2006)

If the security access service is implemented by CANdesc (see the service handler on the service 0x27 instances), you can set here the level specific attributes, like attempts to start the delay time, delay time on power on, etc.

Caution

It is OEM specific property whether the security access parameters will be evaluated security level specific or not. In case the security access service specification of the concrete OEM requires only global configuration of these options, the code generator will calculate the maximum value over all levels for each parameter and this value will be used by the service implementation in CANdesc.

Example: Level 1 has “Attempt Counter” = 1, and Level 2 has for the same parameter = 3. CANdesc will use then for “Attempt Counter” = 3 for all security levels.

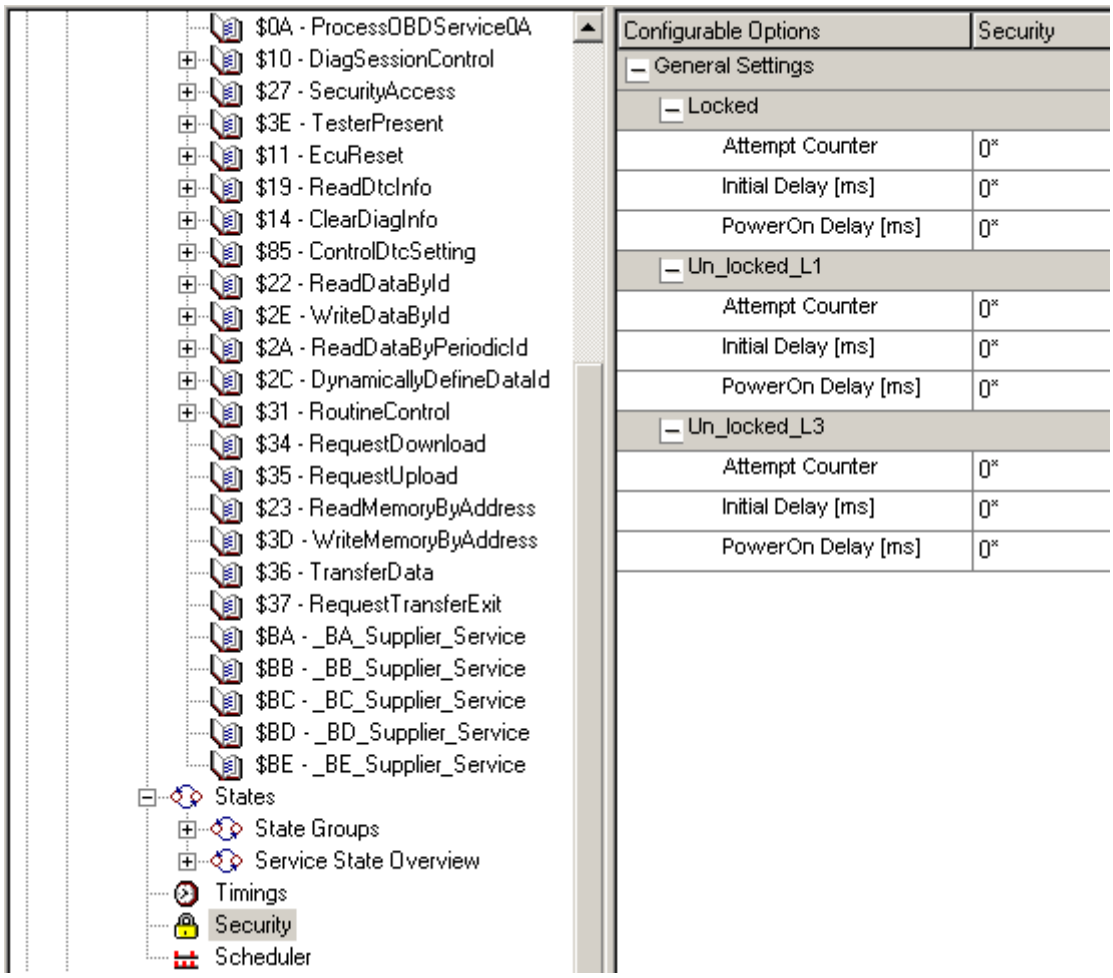


Figure 6-12 GENy CANdesc security access parameters

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
Attempt Counter	Only if the SecurityAccess state group is available	Integer	**0** 1..255	Specifies the maximum number of failed attempts to unlock the ECU. If this number is reached, a delay for the next security access try will be inserted. If a non-zero value is entered, the delay time must be set to a non-zero value too. Note: This parameter has only effect only if the SecurityAccess service is handled by CANdesc.
Initial Delay [ms]	Only if the SecurityAccess state group is available	Integer	**0** 1..65535	Specifies the delay time after the maximum retry attempt count has been reached. If a non-zero value is entered, the

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
				attempt count must be set to a non-zero value too. Note: This parameter has only effect only if the SecurityAccess service is handled by CANdesc.
PowerOn Delay [ms]	Only if the SecurityAccess state group is available	Integer	**0** 1..65535	Specifies the delay time at power on. If a non-zero value is entered, the delay time must be set to a non-zero value too. Note: This parameter has only effect only if the SecurityAccess service is handled by CANdesc.

6.2.5 Security Access Settings (UDS2012)

Due to the new features in CANdesc UDS2012, the configuration of the security levels in GENy has changed.

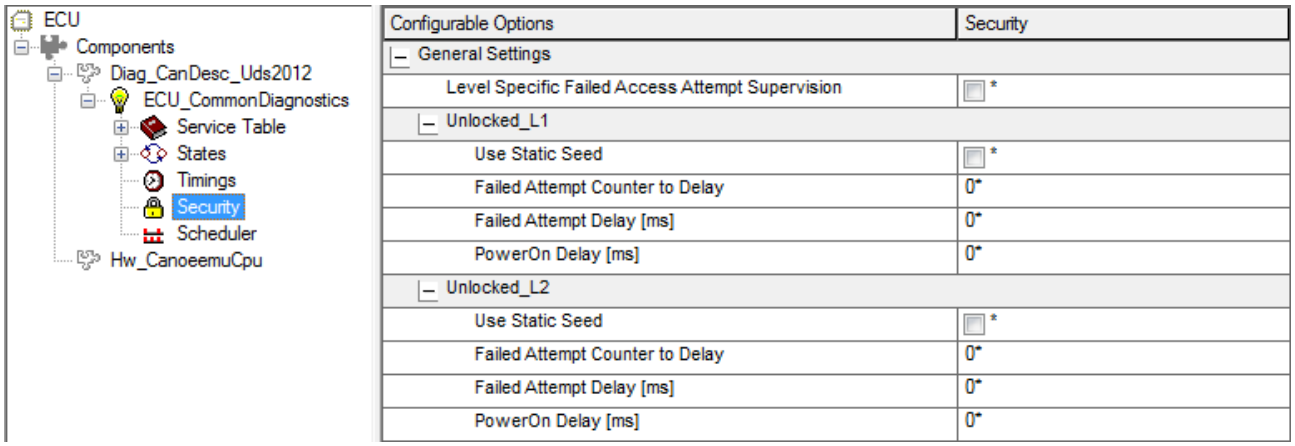


Figure 6-13 Security settings in GENy

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
Level Specific Failed Access Attempt Supervision	Only if the SecurityAccess state group is available	Boolean	**False** True	Switch to select whether a global false attempt counter and delay timer for all security levels shall be used (false) or if each level has its own false attempt counter and delay timer (true).
Use Static Seed	Only if the SecurityAccess state group is	Boolean	**False** True	For each level can be selected if a static seed is used (true) or not (false). Static seed means that

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
	available			CANdesc stores the seed and re-uses the seed in a positive response to a seed request for that level, until the level is unlocked.
Failed Attempt Counter to Delay	Only if the SecurityAccess state group is available	Integer	**Value imported from the Cdd file.** 0..65535	The number of failed security unlock attempts allowed before a delay is imposed between attempts.
Failed Attempt Delay [ms]	Only if the SecurityAccess state group is available	Integer	**Value imported from the Cdd file.** 0..65535	The delay time in ms which is imposed if the Failed Attempt Counter limit has been reached. Further security access attempts are discarded, until the delay has expired.
PowerOn Delay [ms]	Only if the SecurityAccess state group is available	Integer	**Value imported from the Cdd file.** 0..65535	The delay time in ms which is imposed when the ECU is powered on. Requests to unlock the security level are declined until the delay has expired.

6.2.6 Scheduler Settings

If the ECU shall support the periodic data reading service, the following settings are relevant and shall be setup to match the ECU performance and RAM resource availability.

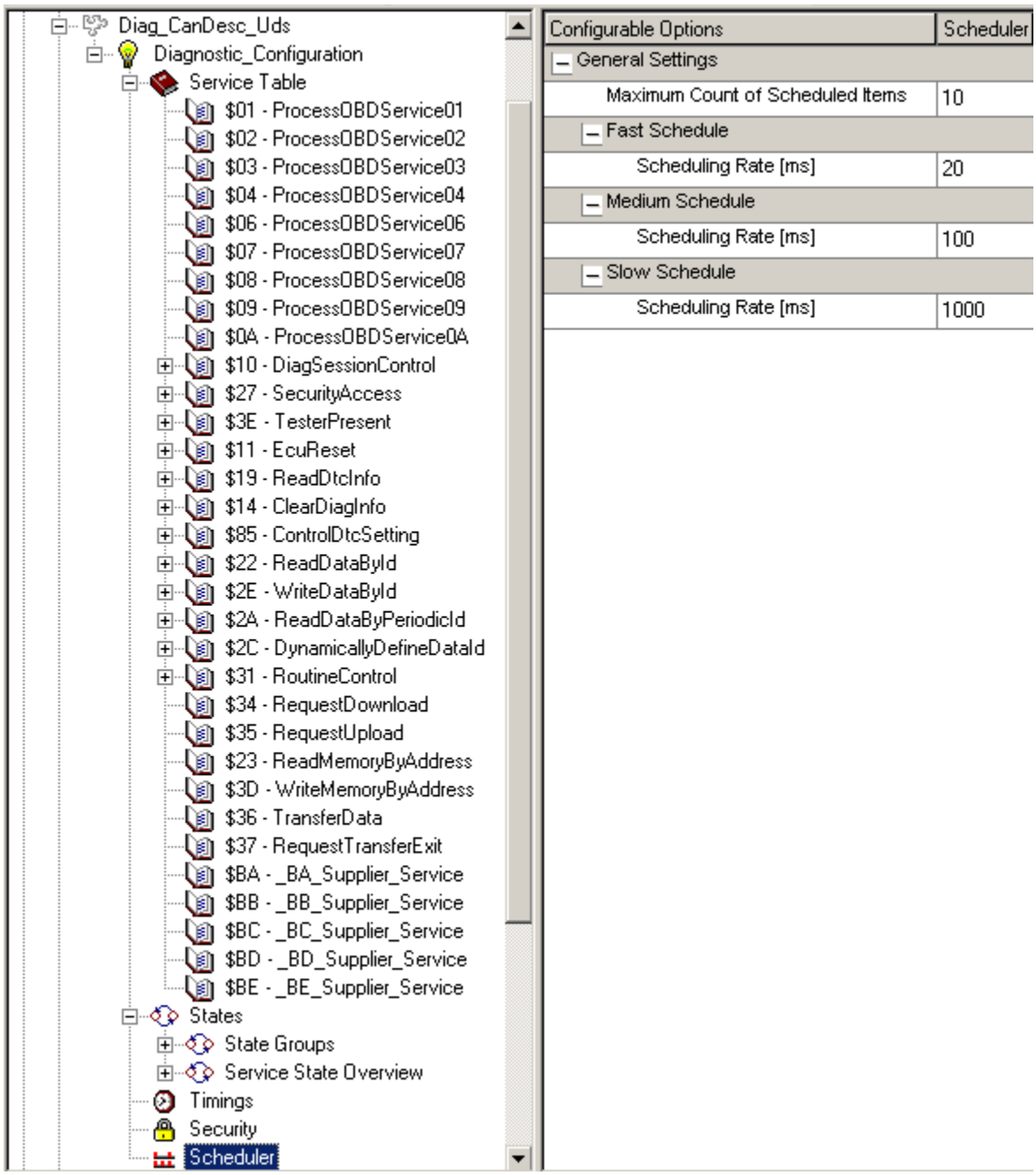


Figure 6-14 GENy CANdesc scheduler parameters

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
Maximum Count of Scheduled Items	Only if the periodic data reading service is available in the ECU configuration.	Integer	**5** 1..255	The maximum number of items that are sent periodically. You can only request at most this number of periodic DIDs, independently per scheduling rate. Example:

Attribute Name	Availability	Value Type	Values The default value is written in bold	Description
				If set up 5 items for scheduling, CANdesc will be able to schedule at most 5 items at fast, 5 items at slow and 5 items at medium rate. Note: If the scheduler size exceeds the total number of available periodic DIDs, CANdesc will automatically reduce the size to the lowest value.
Fast/Medium/Slow Scheduling Rate [ms]	Only if the periodic data reading service is available in the ECU configuration.	Integer	**OEM dependent** 1..65535	Specifies the timings of each scheduling rate that the ECU supports.

7 CANdescBasic Configuration in GENy

As already stated in *6 CANdesc Configuration in GENy* since version 6.00.00, the CANdesc configuration in GENy has been changed. Both CANdesc and CANdescBasic variants share the same GUI and settings representation in GENy. Due to the reduced feature set in CANdescBasic, its GENy GUI provides you correspondingly a reduced configuration option set, covering all of the CANdescBasic requirements.

7.1 Global CANdescBasic Settings

CANdescBasic shares the same global settings as the CANdesc variant (refer to chapter *6.2.1 Global CANdesc Settings*).

Info
CANdescBasic does not support any of the multi identity modes!

7.2 Service Specific Settings

In CANdescBasic, you don't have any more an external diagnostic specification document that shall be imported (like a CDD file). In your software delivery, there is already a prepared diagnostic configuration template that fulfills the concrete OEM and its diagnostic protocol requirements.

Info
In CANdescBasic versions, prior 6.00.00, it was possible to import information, out of a CDD file, whether a service Id is supported or not-supported and any new sessions. In CANdesc 6.00.00 and newer this feature is temporarily disabled, but you still can manually configure these changes.

Since CANdescBasic provides only a Sid view over the diagnostic services, its service specific configuration is performed primarily within the service overview grid in GENy (refer to chapter 0

Service Specific Settings

CANdescBasic also provides a built in support for some of the diagnostic services like CANdesc, but its scope is reduced (due to lack of enough service definition information) only to the most important for diagnostic communication services (e.g. DiagnosticSessionControl, TesterPresent, etc.). You will recognize these services in GENy as described in chapter *6.2.2.2 Predefined (implemented) Services in CANdesc*.

7.3 Timing Settings

The configuration aspect of the CANdescBasic timings settings is the same as described in *6.2.3 Timing Settings*, with the difference, that here there is no CDD file but a predefined template.

7.4 Diagnostic State Configuration

CANdescBasic has a built in support only for the diagnostic session states. All other states like SecurityAccess and ECU specific service execution conditions shall be implemented by the application.

The supplied CANdescBasic template already includes all mandatory session, specified by the concrete OEM. If some additional sessions needed, you can add them in GENy as shown below:

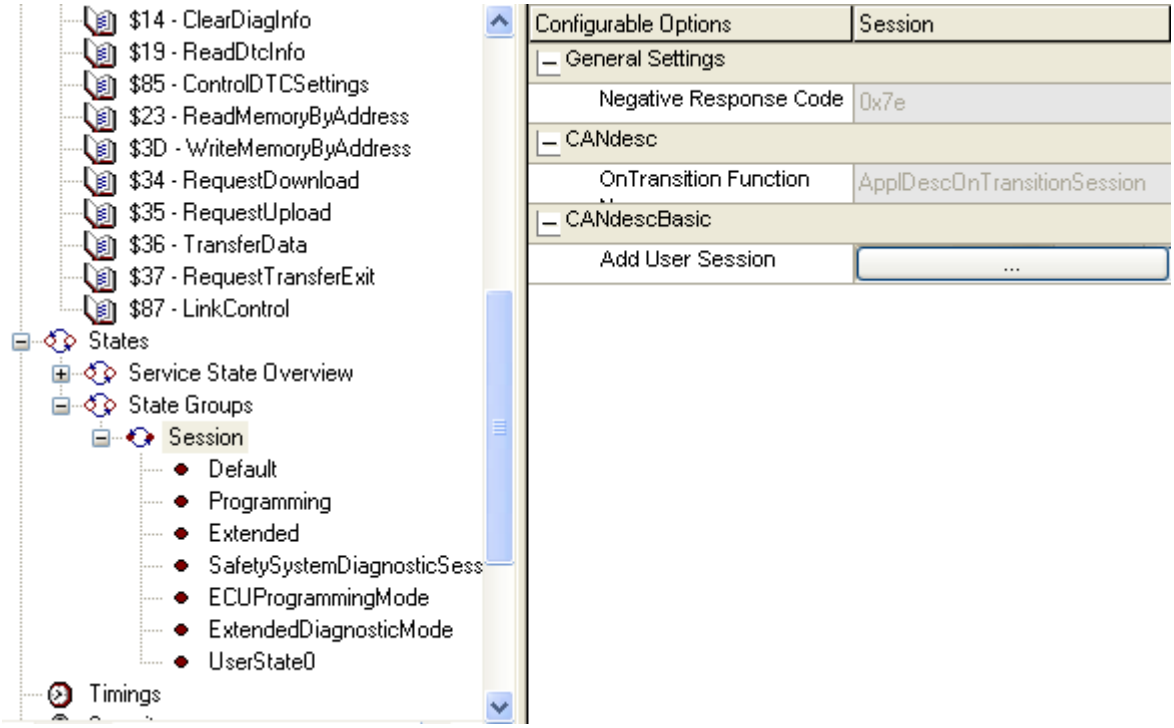


Figure 7-1 CANdescBasic add a user session

Info

For any session added by you (user sessions), GENy automatically creates all session transitions, required by the concrete diagnostic protocol (e.g. UDS, KWP2000).

Examples:

Service 0x10:

<AllExistingSessions>-><NewSsession>,

<NewSession>-><NewSession>

Service 0x20:

<NewSession>-><DefaultSession>

Caution

The allowed session Ids are protocol dependent. For example: on UDS you can not specify user sessions with Ids greater than 0x7F. On KWP2000 any value is acceptable for session Id.

The session Id must be a unique value among all sessions, supported by your ECU.

For the user defined session, you can any time change their name, session or completely remove them:

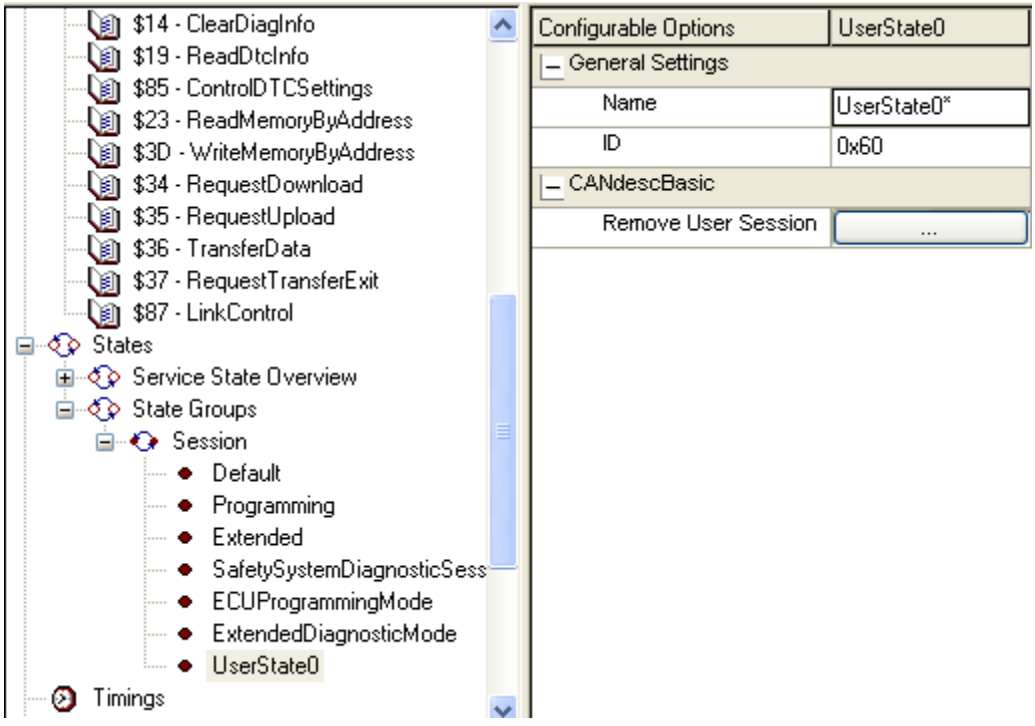


Figure 7-2 CANdescBasic change user session name, id or completely delete user session

Once a user session has been added, you can configure for each service whether it shall be supported or not in the new session. You can do this configuration either on the service overview grid, or if there are some service that have sub-services, for each sub-service. The pictures below show each of the service level configuration views.

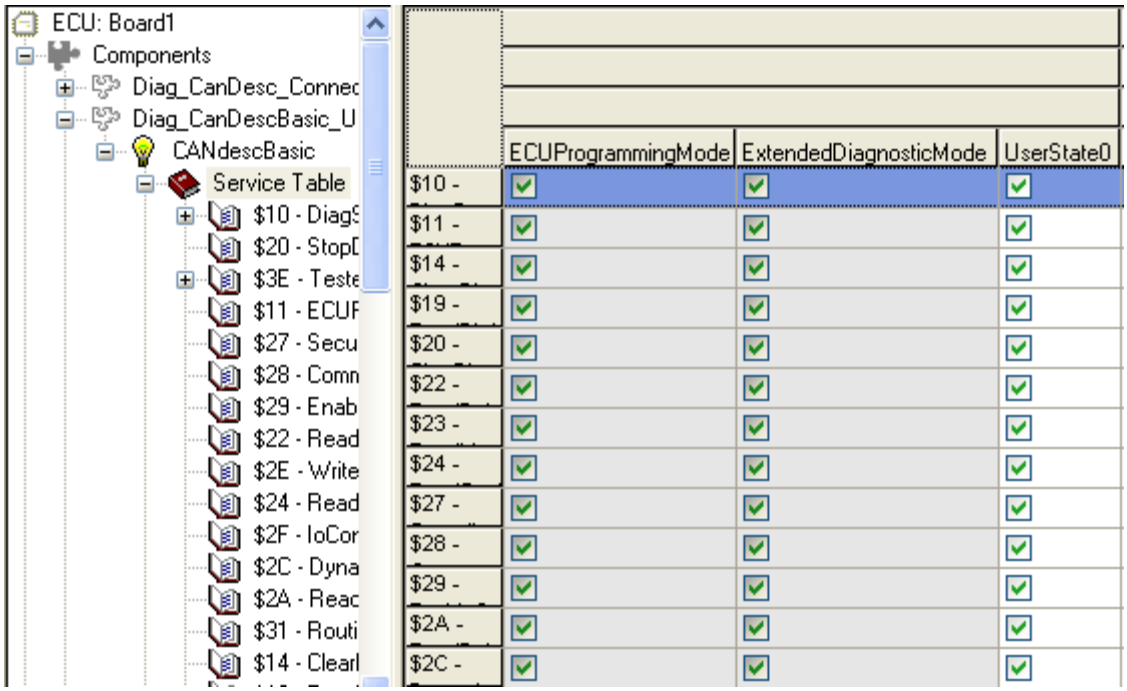


Figure 7-3 CANdescBasic session configuration at service overview

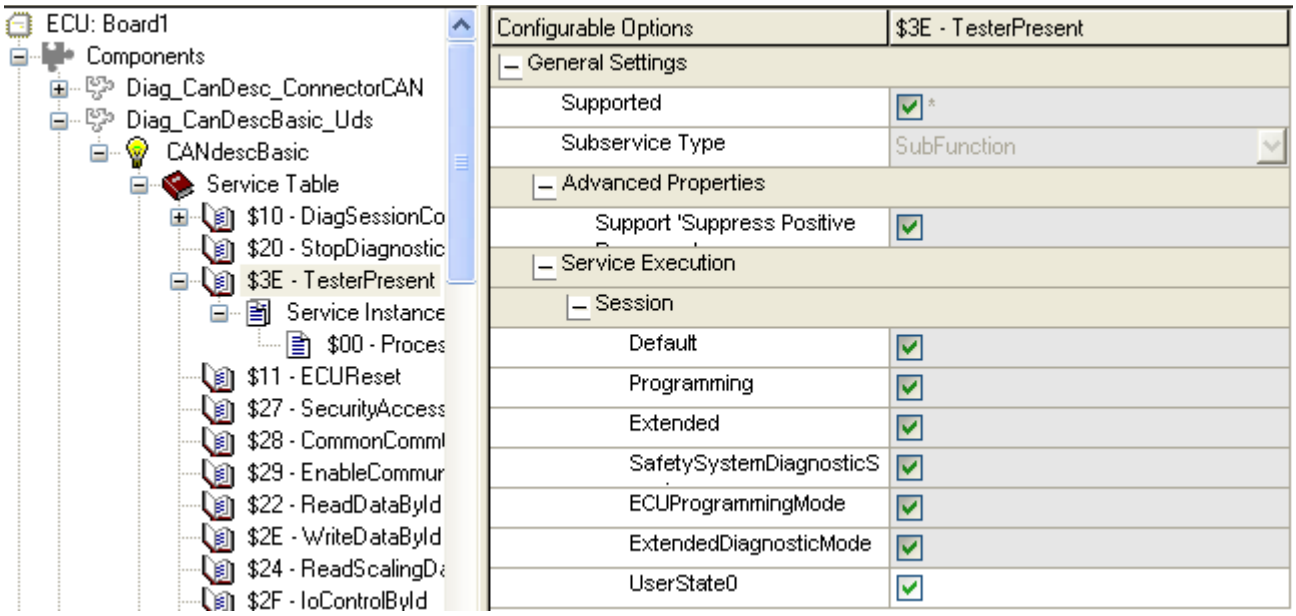


Figure 7-4 CANdescBasic session configuration at service Id level

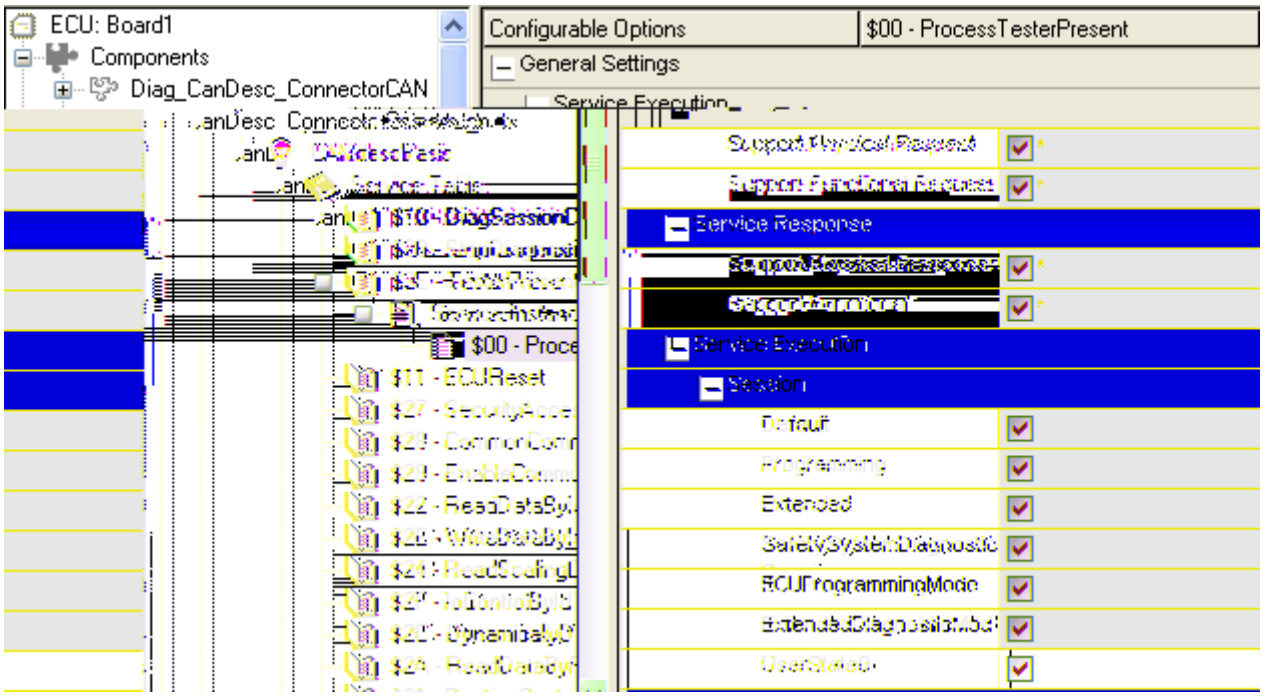


Figure 7-5 CANdescBasic session configuration at sub-service level

8 Multi Identity Support

CANdesc allows you to use multiple diagnostic configuration sets – a use case where the ECU always communicates over the same connection, but shall implement different functionality depending on some hardware (jumper) setting.

All supported configuration sets are described in the following chapters.

Info
Please note:
The multi identity feature of CANdesc is:
- firstly supported in CANdesc 6.00.00;
- not supported at all in the CANdescBasic variant.

8.1 Single Identity Mode

CANdesc has a static configuration set – once all services and communication connections are configured, and the program code is flashed into the ECU there are no more configuration changes possible.

8.1.1.1 Configuration in CANdela

You need just to prepare the corresponding CDD variant for your ECU configuration in CANdelaStudio.

8.1.1.2 Configuration in GENy

Import the CDD file and the corresponding variant in GENy (refer to *chapter 6.2 Step Two – ECU Diagnostic Configuration in GENy* for details).

8.2 VSG Mode

The VSG mode is a special multi identity mode, which has the following characteristics:

- Allows to support multiple diagnostic configuration variants – each variant reflects a VSG from the imported CDD file, and additionally there is a base variant that contains all services that does not belong to any VSG.
- One or several configuration variants can be simultaneously activated during the ECU initialization. The base variant is always active.

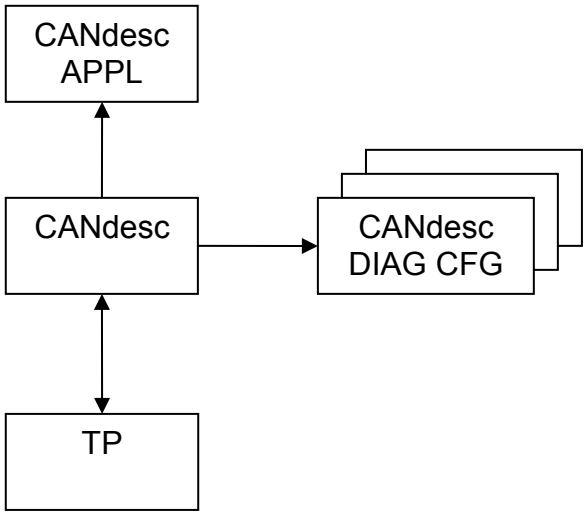


Figure 8-1 CANdesc multi identity mode

CANdesc will be initialized with the base variant at ECU start up sequence. If required, additional variant(s) can be activated by the application (please refer to chapter *12.6.2 Multi Variant Configuration Functions* for more information about the variant initialization).

8.2.1 Implementation Limitations

In order to generate the correct NRC for a requested service Id (e.g. 0x7F (ServiceNotSupprtedInActiveSession), CANdesc considers all of its sub-services diagnostic session specific execution precondition and calculates a diagnostic session filter for the SID. In case of a multi-identity such a calculation shall be made for all of the diagnostic configuration variants, which will cost a lot of ROM resources.

In order to keep CANdesc ROM resources as low as possible the service Id specific session filtering is created considering the superset of all sub-services it contains, independently of their configuration affiliation. Depending on the active configuration set in the ECU, this limitation can lead to the following effect:

A requested service will be responded with the NRC 0x12 (SubfunctionNotSupported) or 0x31(requestOutOfRange), depending on if it has a sub-function or not, instead of the NRC 0x7F. Such a configuration could be for example:

Service 0x22 (ReadDataByIdentifier) supports only two DIDs:

0xF100 - supported only in the default diagnostic sessions and available only in variant 1;

0xF101 – supported only in a non-default session and available only in variant 2.

CANdesc will summarize in this case, that service 0x22 is allowed in any diagnostic session since there is at least one DID supported in at least one of each session.

Now let’s assume the ECU is powered up with active variant 2. If the client sends a request 0x22 0xF100 while in the default diagnostic session, CANdesc will respond with

the NRC 0x31 (DID not supported), instead of the 0x7F (none of the DIDs in the active configuration is executable in the default session -> the service Id itself is not executable in the session -> NRC 0x7F would be expected).

8.2.2 Configuration in CANdela

- If multiple diagnostic configuration sets shall be selectable in CANdesc, you will need a CDD with several VSGs where each describes a diagnostic configuration set.

Caution

CANdesc supports the multiple diagnostic configurations only on service/sub-service availability level. Therefore the following limitations must be considered while creating the separate CDD files resp. CANdela variants for CANdesc:

- A service can be completely deactivated within a VSG;
- A sub-service (e.g. DID, sub-function, etc.) can be completely deactivated within a VSG;
- If a service exist in multiple VSGs, then it must have exactly the same properties
  - Execution pre-conditions (e.g. diagnostic session, security access, etc.)
  - Support of SPRMIB
  - Addressing mode (physical/function)
  - Response behavior (response on physical/function request)
- If a sub-service exist in multiple VSGs, then it must have exactly the same properties
  - Execution pre-conditions (e.g. diagnostic session, security access, etc.), resp. trigger of state transitions.
  - Addressing mode (physical/function)
  - Response behavior (response on physical/function request)
  - Protocol information semantic (sub-function, identifier, etc.)
  - Request resp. response content must be identical – same data structure, data types, and constant value (if any available)
- Service 0x31 (RoutineControlByIdentifier) specifics
  - The multi-identity varying is allowed only on RID level. If a RID is supported in multiple variants, then the sub-functions supported by this RID must be the same (i.e. it is not allowed to have one variant with only “start” sub-function and one with “start and stop” for the one and same RID).
- Service 0x2F (IoControlByIdentifier) specifics
  - The multi-identity varying is allowed only on DID level. If a DID is supported in multiple variants, then the control options supported by this DID must be the same (i.e. it is not allowed to have one variant with only “ShortTermAdjustment” and one with “ShortTerm-Adjustment and ReturnControlToEcu” for the one and same DID).

If at least one of the above requirements is not fulfilled, the variant that violates the rule will not be imported.

8.2.3 Configuration in CANdela

Please follow the steps below on how to configure VSG in CANdelaStudio.

1. Defining all available VSGs for the concrete ECU.

In CANdelaStudio, select the Vehichle System Groups view and add all necessary VSGs into the VSG pool.

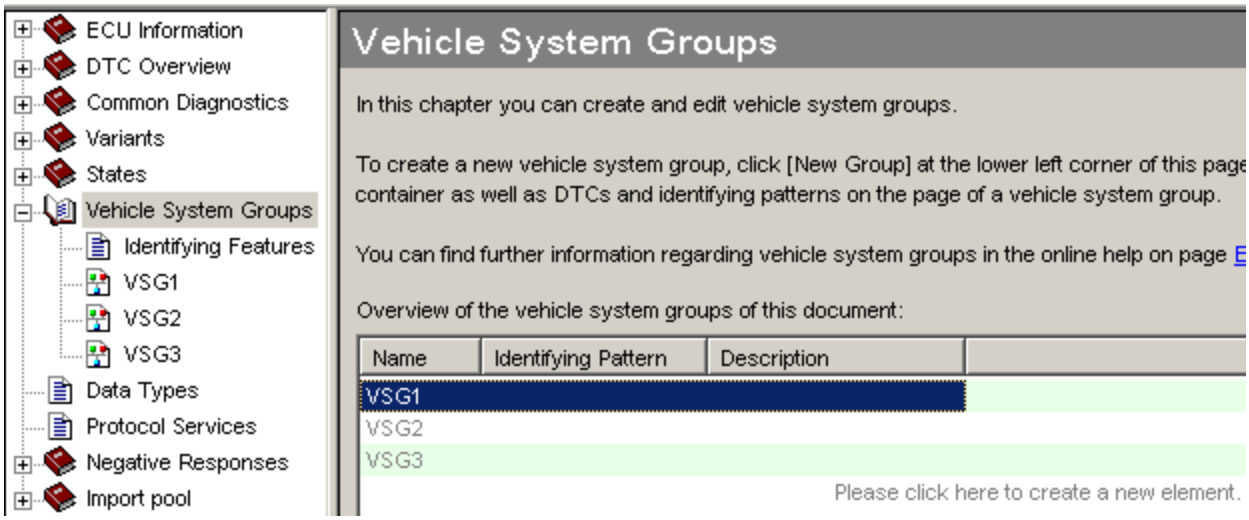


Figure 8-2 Defining VSGs in CANdelaStudio

The name of the created VSG will be used later by CANdesc for the diagnostic configuration constants that the CANdesc application shall use during the configuration activation phase (refer to chapter *12.6.2 Multi Variant Configuration Functions*).

Once all of the required VSGs are created, you can start with the service to VSG assignment.

2. Service to VSG assignment

Using CANdelaStudio you can assign any diagnostic instance to none, one or multiple VSGs. Those services that do belong to a diagnostic instance without a VSG assignment will be considered as services of the base variant (services that are always available).

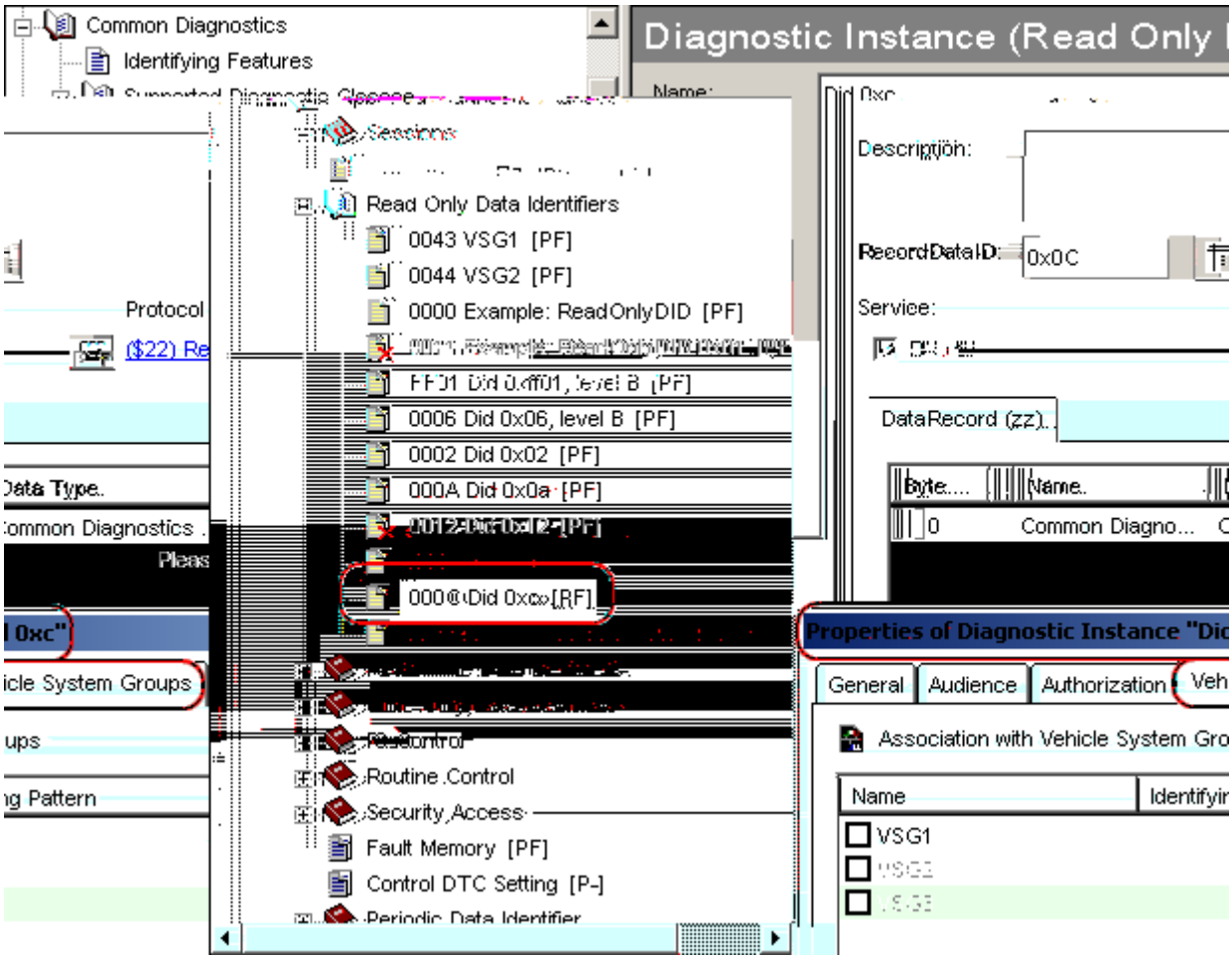


Figure 8-3 Setting a VSG for service in CANdelaStudio

8.2.4 Configuration in GENy

In order to put GENy into VSG mode, you have to select it on the CANdesc component root. Please refer to the chapter *6.2.1 Global CANdesc Settings* for details about the variant selection option.

Now import the CDD file, containing the VSGs in GENy as described in chapter *6.1 Step One – Importing an ECU Diagnostic Description*. That is all.

8.3 Multi Identity Mode

Multi Identy Mode is not supported by CANdesc.

9 Diagnostic Service Implementation Specifics

9.1 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 9.1.2 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 (9.1.3 Multiple PID mode) is showing the multiple PID processing.

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

9.1.1 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.

9.1.2 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.

9.1.2.1 Sending a positive response using linear buffer access

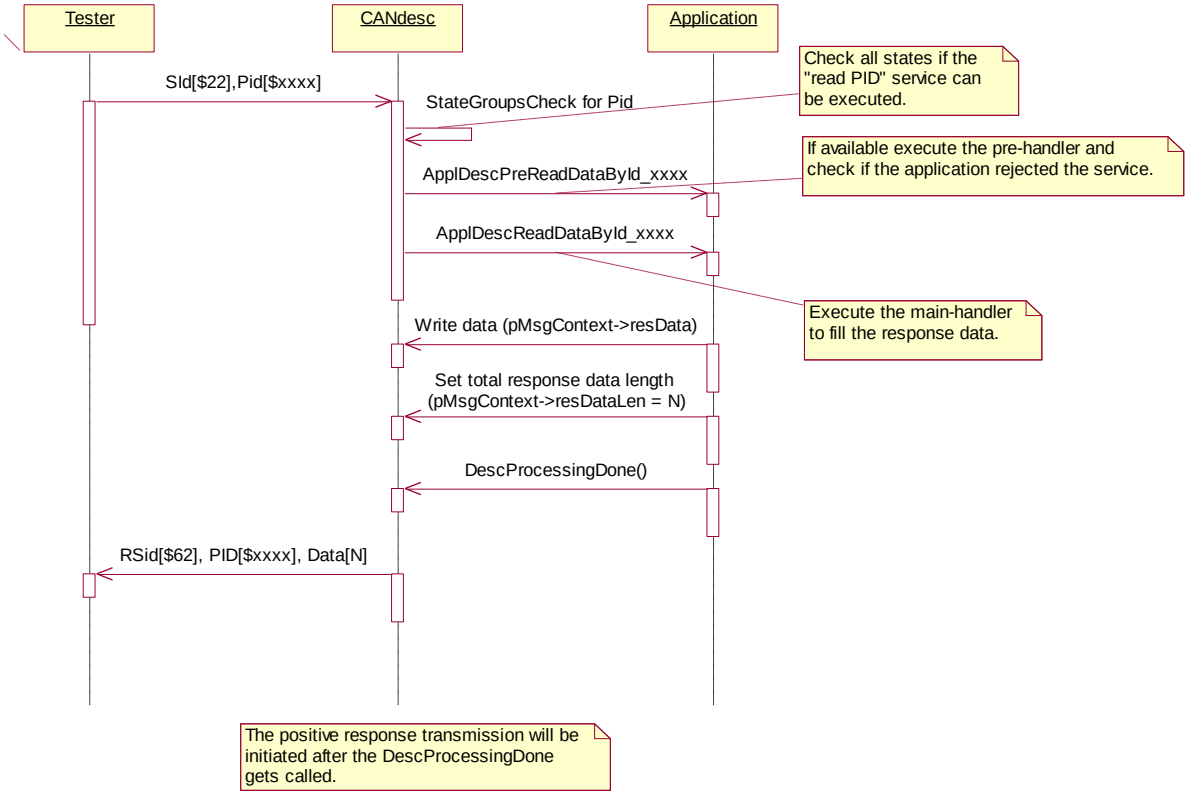


Figure 9-1: Linearly written positive response on single PID request

9.1.2.2 Sending a positive response using ring buffer access

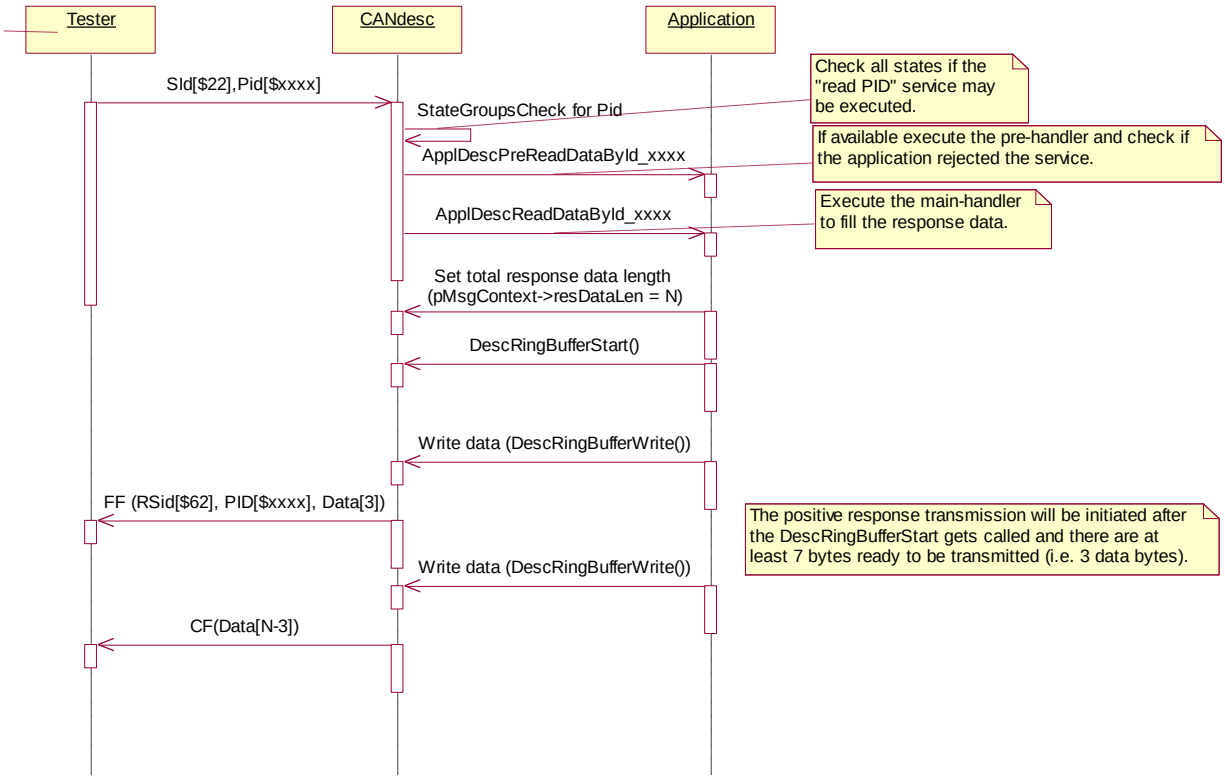


Figure 9-2: “On the fly” response data writing.

9.1.2.3 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 9.1.3.2 “Ring buffer active configuration” for the recommended negative response handling.

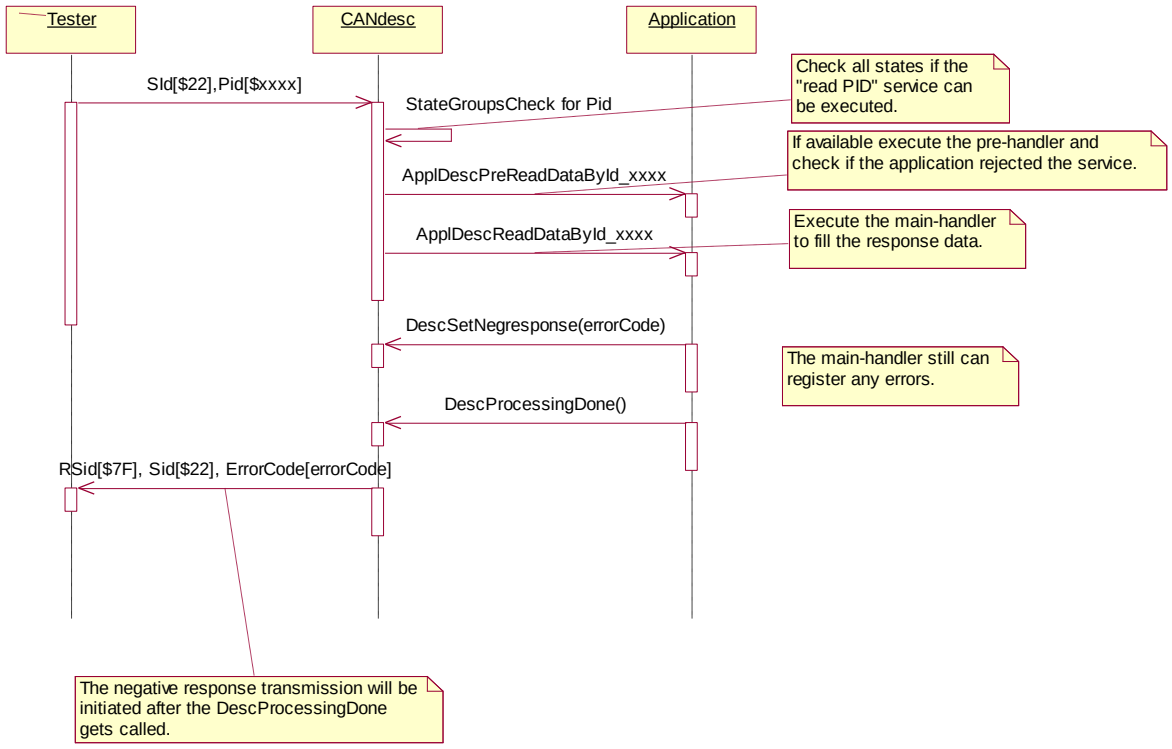


Figure 9-3: Negative response on single PID

9.1.3 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.:

9.1.3.1 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.

9.1.3.1.1 Sending a positive response

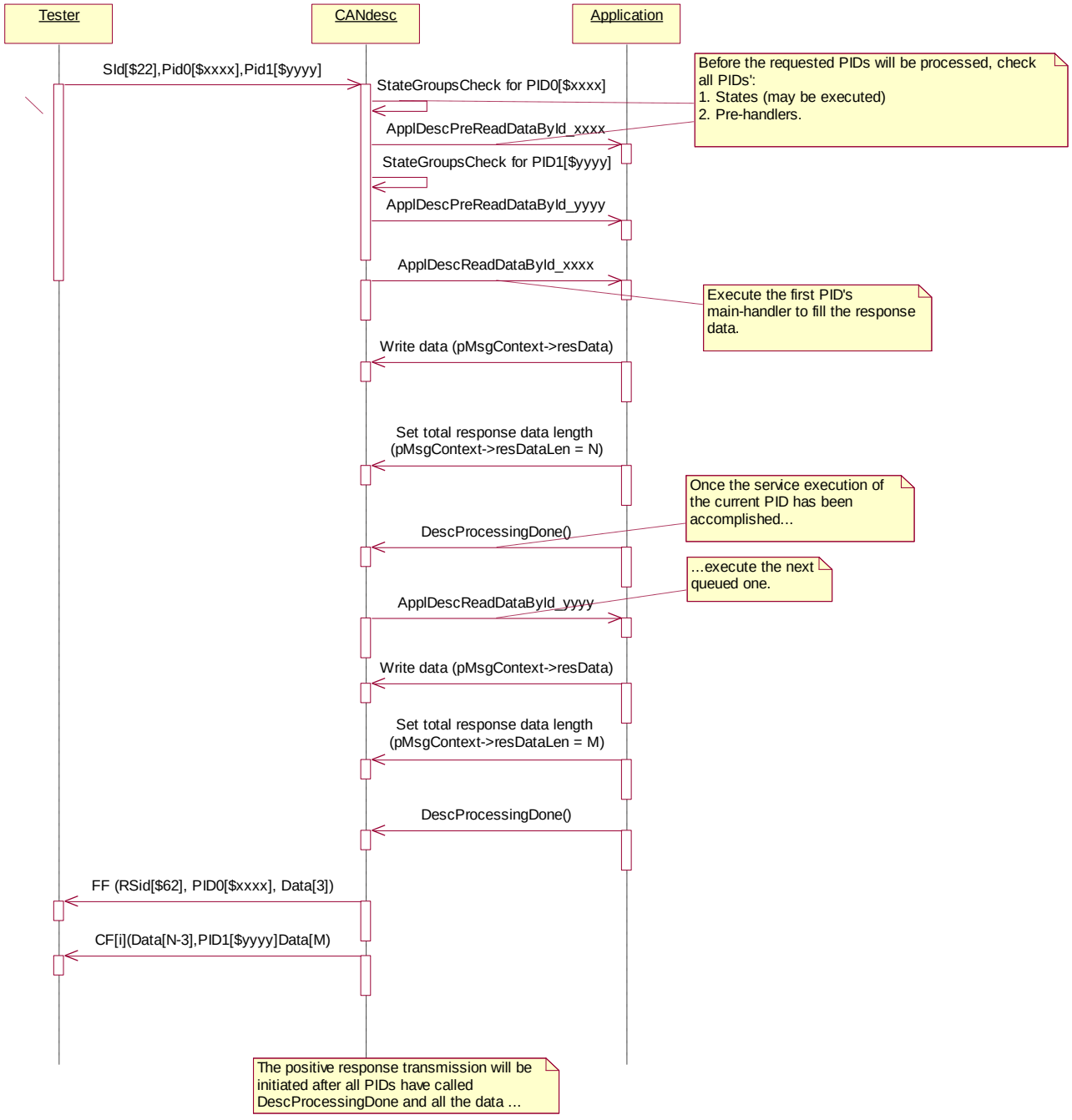


Figure 9-4: Linearly written positive response on multiple PIDs (global ring buffer option is off)

9.1.3.1.2 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.

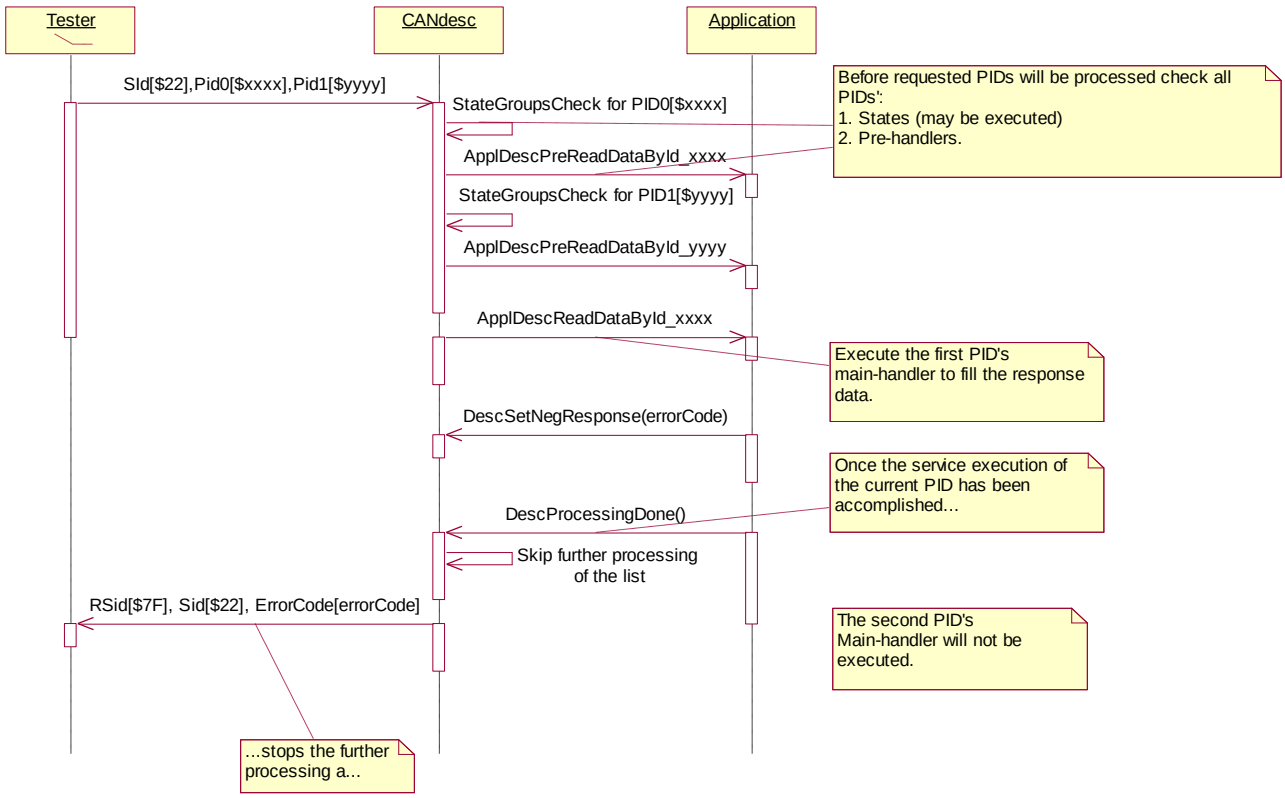


Figure 9-5: Negative response on multiple PIDs (global ring buffer option is off)

9.1.3.2 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.

9.1.3.2.1 Sending a positive response

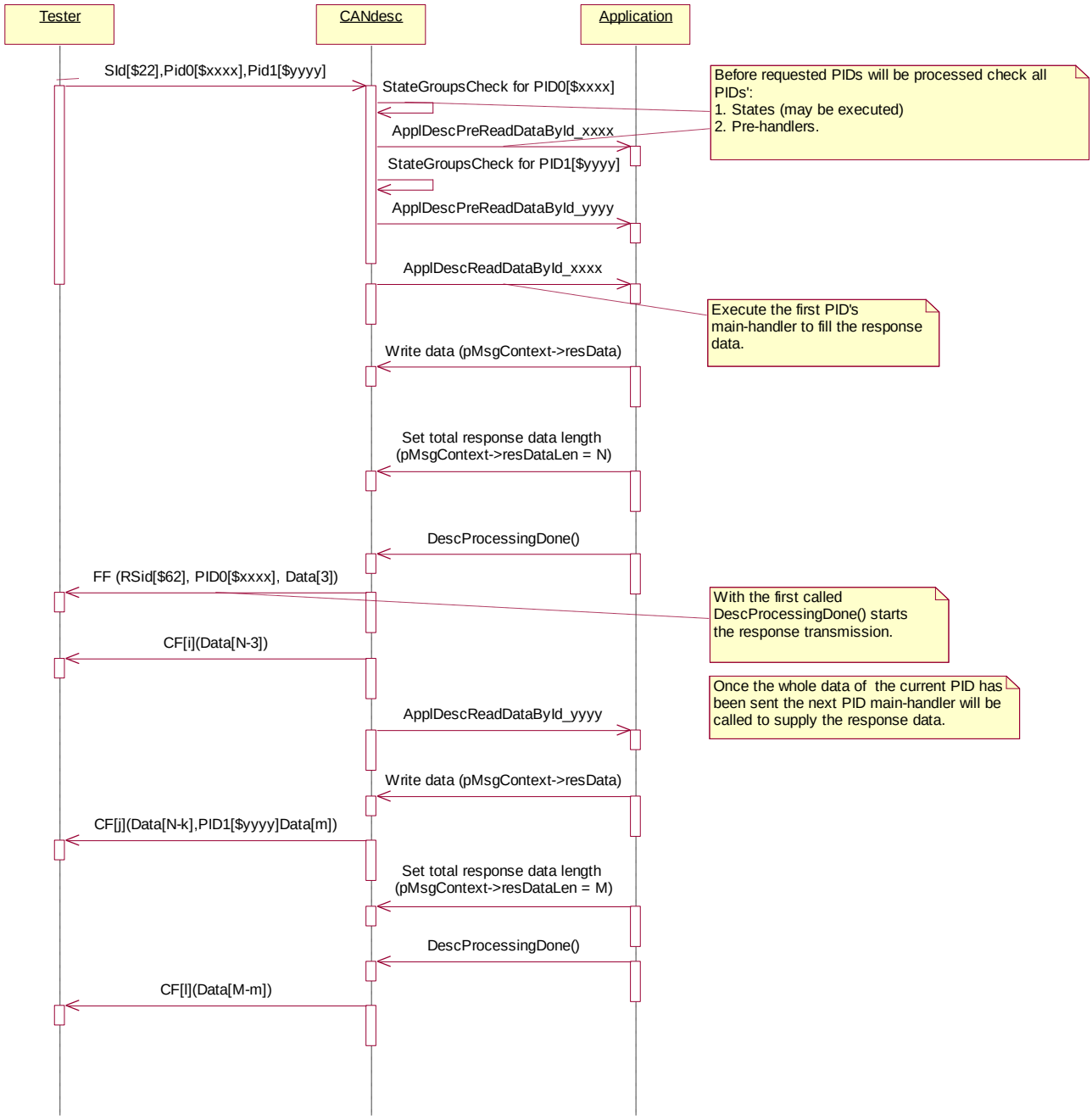


Figure 9-6: Linearly written response data on multiple PIDs (global ring buffer option is on)

9.1.3.2.2 Sending a negative response

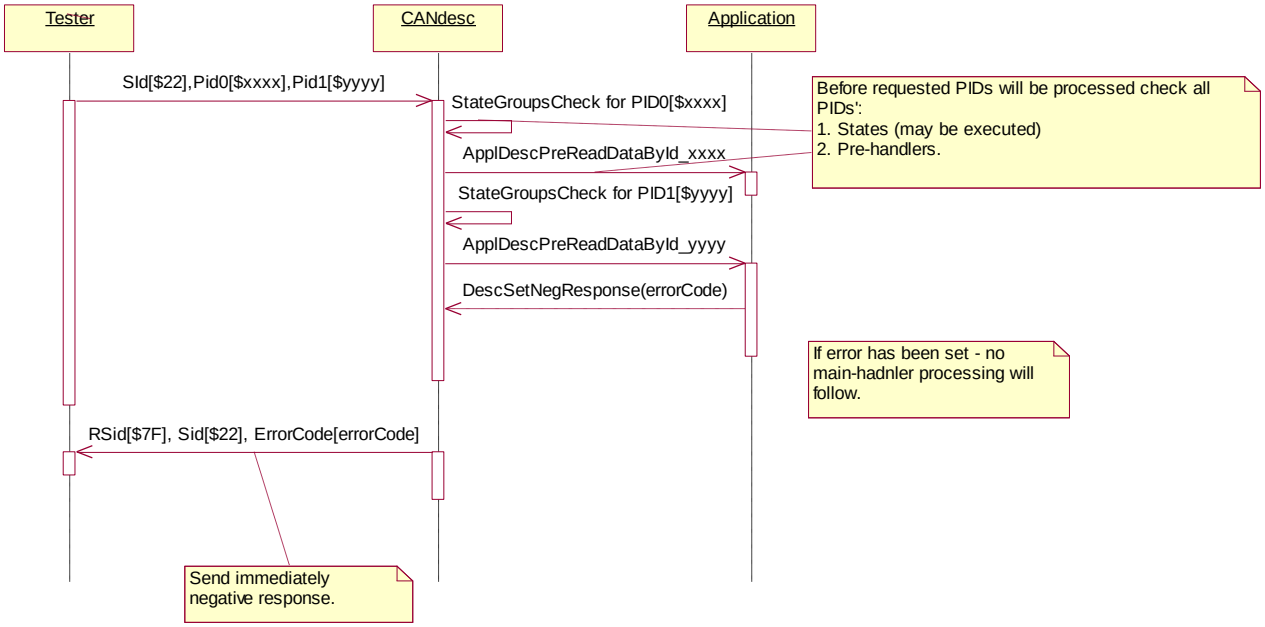


Figure 9-7: Negative response on multiple PIDs (global ring buffer option is on)

9.1.3.2.3 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:

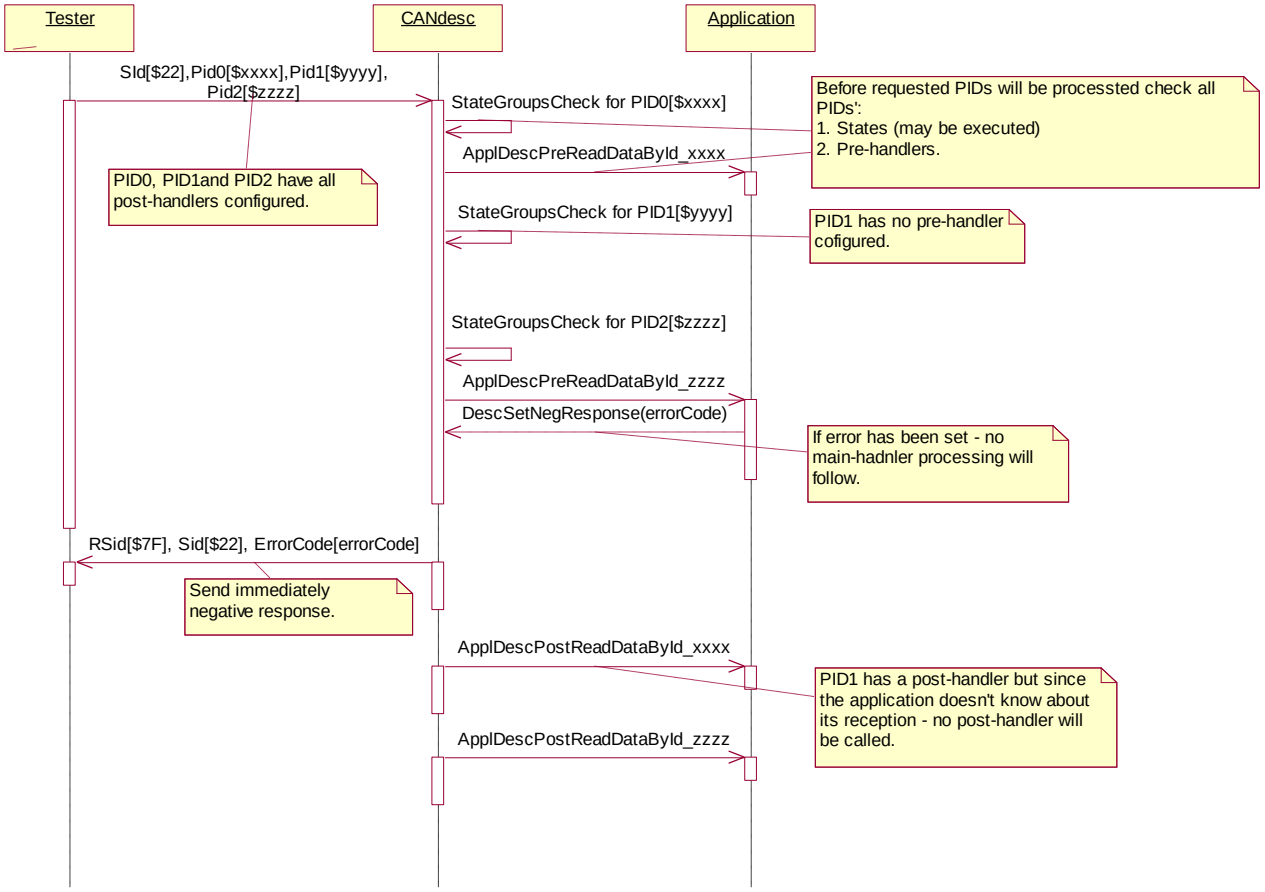


Figure 9-8: Post-Handler execution sequence.

9.2 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/

9.2.1 Feature set

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

Subfunction Name	Hex Value
defineByIdentifier	01
defineByMemoryAddress	02
clearDynamicallyDefinedDataIdentifier	03

9.2.2 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 12.6.13 “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 12.6.13.2 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.

9.2.3 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.

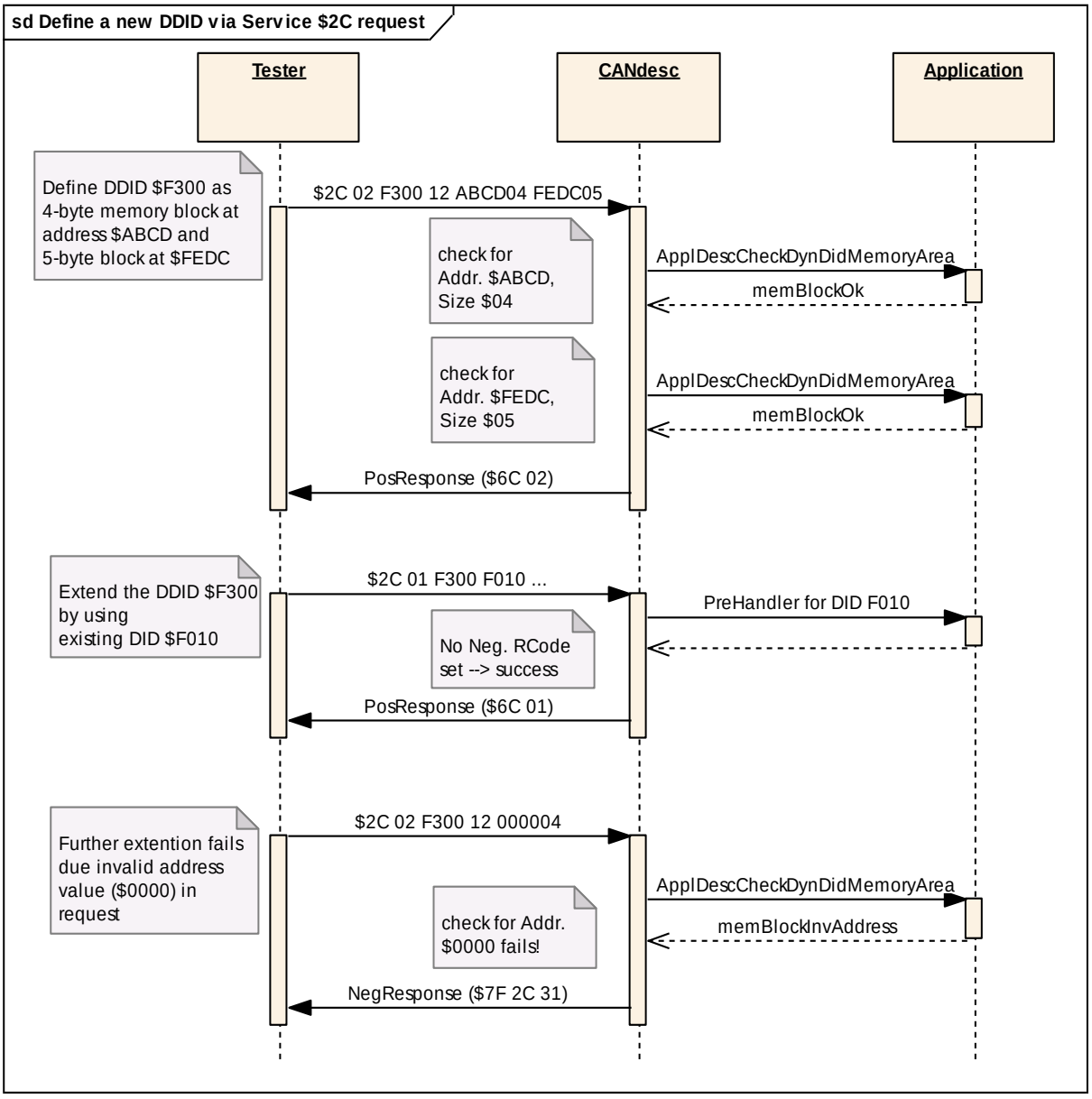


Figure 9-9: Defining a DDID.

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.

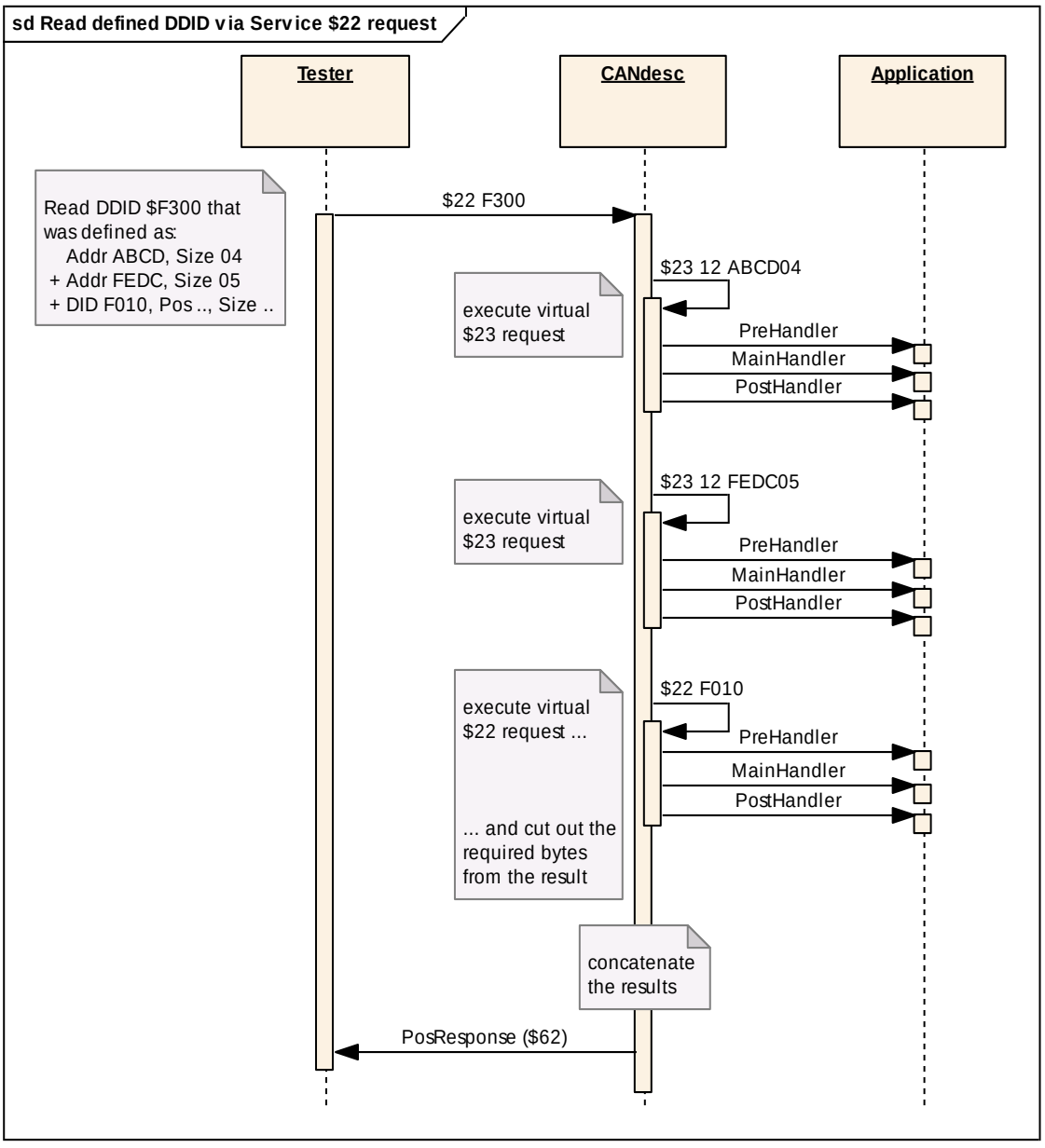


Figure 9-10: Reading a DDID.

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!

9.3 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 (12.6.14.1)

ApplDescWriteMemoryByAddress (12.6.14.2)

9.3.1 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.

9.3.2 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.

9.3.3 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 12.6.8.1)’ can not be used with the memory access callbacks.

Instead, a new API call ‘DescStartMemByAddrRepeatedCall (see 12.6.8.2)’ has been added.

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

10 Generic Processing Notifications

If CANdesc UDS2012 is used, the feature “Generic Processing Notifications” is provided. Upon activating this feature, CANdesc will notify the application when the processing of a request starts and ends. Thereby, the notification mechanism is two-staged. On each stage there are two application callbacks, one indication and one confirmation callback. On the first stage “Manufacturer Notification Support”, CANdesc will notify the application right before the processing of a fully received request starts, by calling the function *ApplDescManufacturerIndication()*. When the processing of the request has been finished, the response has been sent and all PostHandlers were called, CANdesc notifies the application again by calling the function *ApplDescManufacturerConfirmation()*.
The application callbacks of the second stage “Supplier Notification Support” are named accordingly *ApplDescSupplierIndication()* and *ApplDescSupplierConfirmation()*. The indication callback is called by CANdesc after it has verified that the requested service is supported in the active session, security state and user states. The confirmation callback is also called after the response has been sent, and all PostHandlers were called, but right before the call to *ApplDescManufacturerConfirmation()*. Thus, the manufacturer and supplier callbacks are called in a nested way. Figure 3-1 illustrates the order of the notification callbacks related to the processing of a service request.

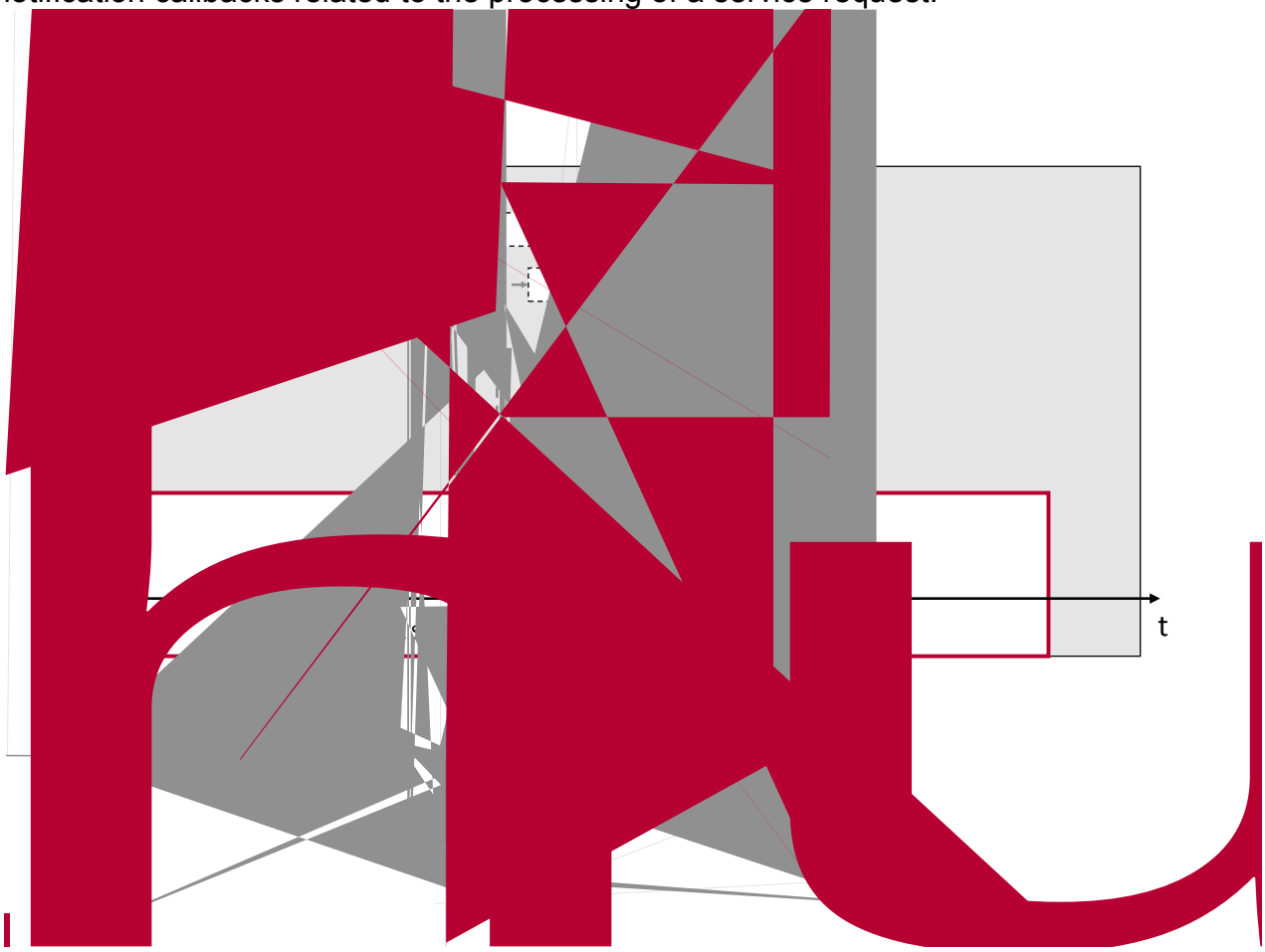


Figure 10-1 Call order of Manufacturer- and Supplier-Notficiation

10.1 Using dynamically defined data Identifier

The Service DynamicallyDefineDataIdentifier allows the definition of data identifiers with other data identifiers or memory areas. These DDIDs can be read via service ReadDataByIdentifier. When reading a DDID, for each source element a virtual request is processed by CANdesc to get the information for this source element from the application(see chapter 9.2). Because CANdesc processes the virtual requests equal to normal requests, the notification functions will not only be called for the $22 request containing the DDID, but also for each virtual request. The application has to consider these additional calls, in case a DDID is requested.

Figure 10-2 shows an example of reading a DDID with service $22.

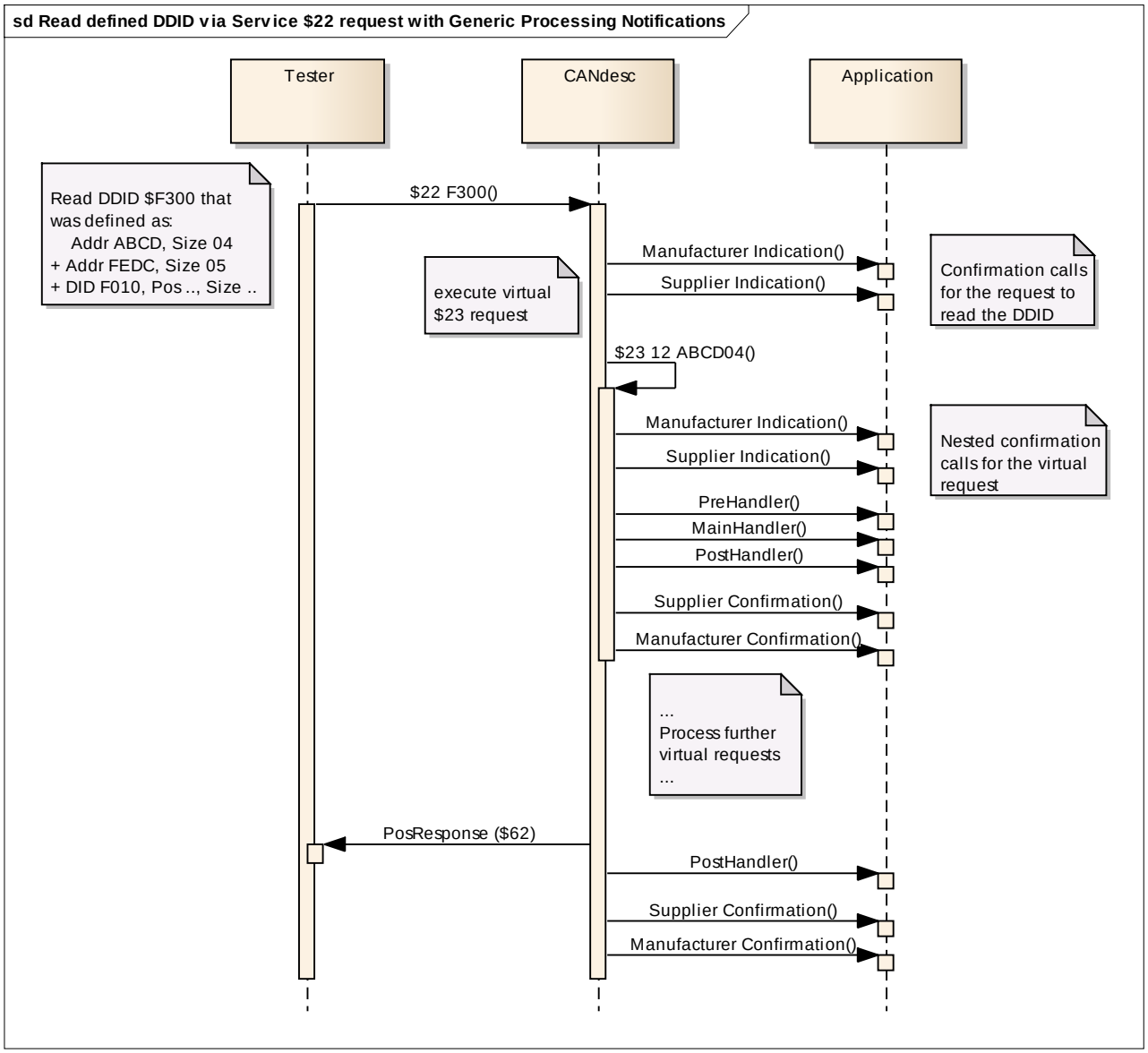


Figure 10-2 Read out a DDID with generic processing notifications

11 Busy Repeat Responder Support (UDS2006 and UDS2012)

Busy Repeat Responder is a feature, that allowes CANdesc to respond to incoming requests during the processing of another request. Such parallel requests are properly received and in the next task cycle of CANdesc responded negatively with NRC BusyRepeatRequest (0x21).

Figure 11-1 illustrates the functionality of the Busy Repeat Responder mechanism. During the processing of Request 1, Requests 2 and 3 from Tester 2 are responded negatively with NRC BusyRepeatRequest. After the processing of request 1 has finished and a positive response has been sent, Request 4 from Tester 2 can be processed properly.

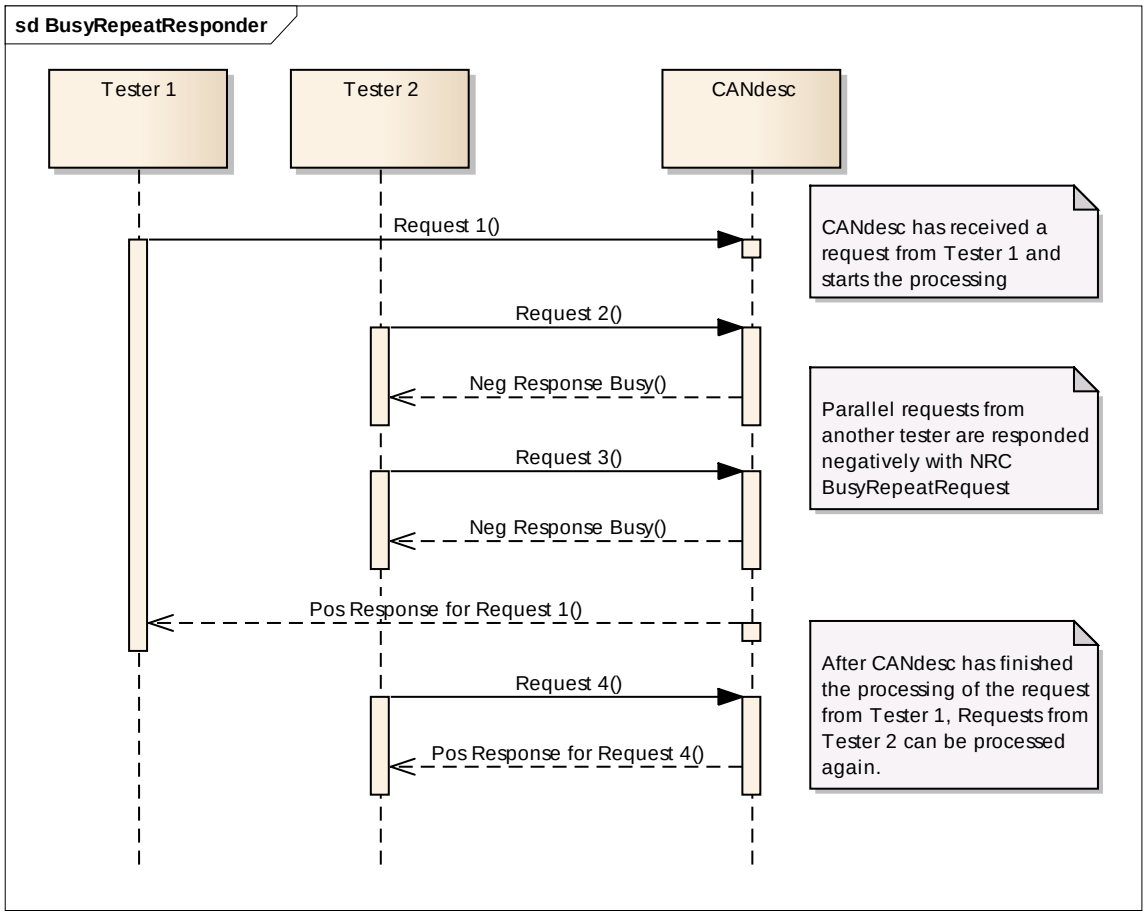


Figure 11-1 Illustration of the feature BusyRepeatResponder

Preconditions that must be fulfilled when using the feature Busy Repeat Responder:

- The TP must be a ISO TP from Vector with TP Class “Dynamic Normal Addressing Multi TP” or “Dynamic Normal Fixed Addressing Multi TP”
- In the TP configuration the feature “Extended API – Overrun Reception” must be active
- In the TP configuration the number of Rx channels and Tx Channels must be > 1
- In case of Dynamic Normal Addressing Multi TP, a dispatcher needs to be implemented in the application (for a detailed description see chapter *13.12*)

Restrictions when using the feature Busy Repeat Responder:

> Only physical parallel requests are responded negatively. Functional parallel requests will NOT get a negative response.

11.1 Configuration in GENy

To activate the feature Busy Repeat Responder use the setting in the CANdesc component root (refer to chapter *6.2.1 Global CANdesc Settings).*

Furthermore, the feature requires additional configuration in the TP component. The feature “Extended API – Overrun Reception” must be enabled. This setting is available in the group “Advanced Configuration”. To be able to receive another request while one is under processing, the “Number of Rx Channels” and “Number of Tx Channels” must be at least two. The number of channels can be configured in the TP Connection Groups:

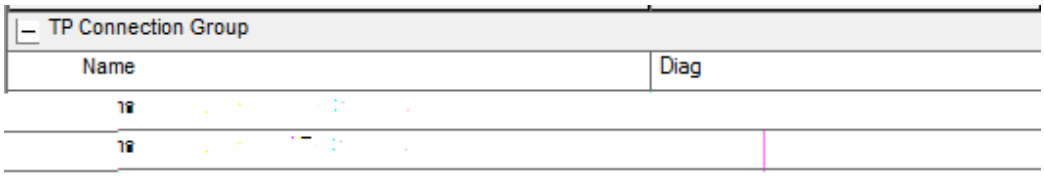


Figure 11-2 Example of the “Number of Rx(Tx) Channels” settings

In case of “Dynamic Normal Addressing Multi TP” a dispatcher needs to be implemented in the application. The description of the GENy configuration to integrate the dispatcher is described in chapter *13.12 …use “Dynamic Normal Addressing Multi TP” with* multiple tester.

12 CANdesc API

12.1 API Categories

12.1.1 Single Context

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

12.1.2 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.

12.2 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 12-1: standard data types

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

12.3 Global Variables

-

12.4 Constants

12.4.1 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 12-2: Version API data

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

12.5 Macros

12.5.1 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.

12.5.1.1 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)

12.5.1.2 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)

12.5.1.3 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.

12.5.1.4 Assembling 32 bit data

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

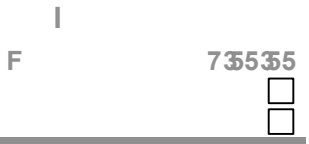
uint32 DescMake32Bit(HiHiByte, HiLoByte, LoHiByte, LoLoByte)

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

12.6 Functions

12.6.1 Administrative Functions

12.6.1.1 DescInitPowerOn()



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	

12.6.1.2 DescInit()

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	

12.6.1.3 DescTask()

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.
-

12.6.1.4 DescStateTask()

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).
-

12.6.1.5 DescTimerTask()

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. ■	

12.6.1.6 DescGetActivityState()

I L F

F 7 3 5 5 3 5

☒

☐

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	

12.6.2 Multi Variant Configuration Functions

12.6.2.1 DescInitConfigVariant()

Prototype	
Single Context and Multi Context	
`void` **`DescInitConfigVariant`** `(DescVariantMask varMask)`	
Parameter	
`varMask`	Contains the VSG(s) that shall be active additionally to the base variant
Return code	
_	-

Functional Description

After CANdesc has been initialized via one of the APIs
DescInitPowerOn
or
DescInit;
the base variant will be only active (refer to the chapter *8 Multi Identity* for more details). If additionally other variants shall be activated, this API shall be called with a parameter value that represents the variants (multiple variants can be OR-ed) that shall be activated.

The variant values that shall be used for building the API parameter value are located in the desc.h file. The naming convention is as follows:
kDescVariant<variant/VSG qualifier>

Pre-conditions

-Multi- variant (VSG) mode is activated for CANdesc.

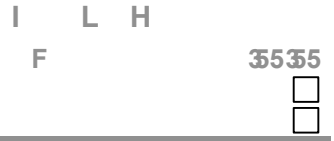
Call context

-

Particularities and Limitations

- Shall not interrupt the DescTask function.
- Best place to call this API is immediately after the CANdesc initialization API-call while the interrupts are still locked.

12.6.2.2 DescGetConfigVariant()



Prototype
Single Context and Multi Context
`DescVariantMask` **`DescGetConfigVariant`** `(void)`

Parameter	
–	

Return code	
`Variant mask`	Represents the bit-mapped value of the currently active variants in the ECU.

Functional Description

This API returns the bit-mapped value of the currently active variants set in CANdesc.

The variant values that shall be used for checking the API return value are located in the desc.h file. The naming convention is as follows:
kDescVariant<variant/VSG qualifier>

Pre-conditions

-Multi- variant (VSG) mode is activated for CANdesc.

Call context

- This API can be called from any call-context.

Particularities and Limitations

- -

12.6.3 Service Functions

12.6.3.1 DescSetNegResponse()

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().	

12.6.3.2 DescProcessingDone()

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

12.6.4 Service callback functions

In CANdesc 6 the naming convention of the service callback function has changed due to standardization reasons. In Table 12-3, the new naming convention can be found. Earlier versions of CANdesc (< 6.0) used always Service-Qualifiers and Instance-Qualifiers from the CDD file. Since CANdesc 6, for Service-Qualifiers always standardized names are used, whereas for Instance-Qualifiers either a standardized name or the name from the CDD file is used. The names of the service callback functions are based on the following pattern:

ApplDesc[Pre|Post]<ServiceQualifier><DiagInstanceQualifier>

When migrating to CANdesc 6 the service callbacks have to be renamed according to the new naming convention.

Service	SubService	Instance-Qualifier	Service-Qualifier
0x10	0x01	Default	StartSession
	0x02	Programming	

Service	SubService	Instance-Qualifier	Service-Qualifier
	0x03	Extended	
0x11	0x01	Hard	EcuReset
	0x02	KeyOffOn	
	0x03	Soft	
	0x04	EnableRapidShutDown	
	0x05	DisableRapidShutDown	
0x14	None	DiagInfo	Clear
0x19	0x01	RNODTCBSM	ReadDtc
	0x02	RDTCBSM	
	0x03	RDTCSSI	
	0x04	RDTCSSBDTC	
	0x05	RDTCSSBRN	
	0x06	RDTCEDRBDN	
	0x07	RNODTCBSMR	
	0x08	RDTCBSMR	
	0x09	RSIODTC	
	0x0A	RSUPDTC	
	0x0B	RFTFDTC	
	0x0C	RFCDTC	
	0x0D	RMRTFDTC	
	0x0E	RMRCDTC	
	0x0F	RMMDTCBSM	
	0x10	RMDEDRBDN	
	0x11	RNOMMDTCBSM	
	0x12	RNOOBDDTCBSM	
	0x13	ROBDDTCBSM	
	0x14	RDTCFDC	
	0x15	RDTCWPS	
	0x16	RDTCRDIDBDN	
	0x41	RWWHOBDNDTCBMR	
	0x42	RWWHOBDDTCBMR	
	0x55	RWWHOBDDTCWPS	
0x22	Any	Instance-Qualifier from CDD	ReadDid
0x23	None	MemoryByAddress	Read
0x24	Any	Instance-Qualifier from CDD	ReadScalingDid
0x27	Odd Id	Instance-Qualifier from CDD	GetSeed
	Even Id		SendKey
0x28	0x00	EnableRxEnableTx	CommCtrl
	0x01	EnableRxDisableTx	

Service	SubService	Instance-Qualifier	Service-Qualifier
	0x02	DisableRxEnableTx	
	0x03	DisableRxDisableTx	
0x2A	0x01	Instance-Qualifier from CDD	ReadDidSlow
	0x02		ReadDidMed
	0x03		ReadDidFast
	0x04		ReadDidStop
0x2C	0x01	Instance-Qualifier from CDD	DynDefineByDid
	0x02		DynDefineByAddr
	0x03		DynDefineClear
0x2E	Any	Instance-Qualifier from CDD	WriteDid
0x2F	0x00	Instance-Qualifier from CDD	IoCtrlRetCtrlToEcu
	0x01		IoCtrlRstToDefault
	0x02		IoCtrlFrzCurrState
	0x03		IoCtrlShortTermAdj
0x31	0x01	Instance-Qualifier from CDD	RtnCtrlStart
	0x02		RtnCtrlStop
	0x03		RtnCtrlReqRes
0x34	None		RequestDownload
0x35	None		RequestUpload
0x36	None		TransferData
0x37	None		RequestTransferExit
0x3D	None	MemoryByAddress	Write
0x3E	0x00	TesterPresent	Send
0x84	None		SecuredDataTransmission
0x85	0x01	Enable	ControlDtcSetting
	0x02	Disable	
0x86	0x00	Stop	Roe
	0x01	OnDtcStatChg	
	0x02	OnTmrInt	
	0x03	OnChgOfDid	
	0x04	ReportActEv	
	0x05	Start	
	0x06	Clear	
	0x07	OnCompOfVal	
	0x40	StStop	
	0x41	StOnDtcStatChg	
	0x42	StOnTmrInt	
	0x43	StOnChgOfDid	
	0x44	StReportActEv	
	0x45	StStart	

Service	SubService	Instance-Qualifier	Service-Qualifier
	0x46	StClear	
	0x47	StOnCompOfVal	
0x87	0x01	VerifyFixedBaudrate	LinkControl
	0x02	VerifySpecificBaudrate	
	0x03	TransitionBaudrate	

Table 12-3 Naming convention of service callback functions in CANdesc 6

12.6.4.1 Service PreHandler

F I A - 0 - CC
F 73553 55
☒

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 Limitations	

12.6.4.2 Service MainHandler

F I A - 0 - C
F 73553 5
☒

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

12.6.4.3 Service PostHandler

F I A - 0 - C F 735535 ☒

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’.	

12.6.5 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).

12.6.5.1 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 12.6.5.2 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 12.6.5.4 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 12.6.5.5 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.

12.6.5.2 ApplDescCheckUserService()

F I H

F 7 3 6 6 3 5 5

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)

12.6.5.3 DescGetServiceId()

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**

12.6.5.4 Generic User Service MainHandler

F I M

F 7366355

Prototype
Single Context
`void` **`ApplDescUserServiceHandler`** `(DescMsgContext* pMsgContext)`
Multi Context
`void` **`ApplDescUserServiceHandler`** `(DescMsgContext* pMsgContext)`

Parameter	
`pMsgContext`	Refer the section 12.6.4.2 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 12.6.4.2 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 12.6.4.2 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 12.6.4.2 Service MainHandler.
- **DescGetServiceId()** may be called here to dispatch the SID of the currently processed user service (refer 12.6.5.3 DescGetServiceId

12.6.5.5 Generic User Service PostHandler

F I M

F 7366355

Prototype	
Single Context	
`void ApplDescPostUserServiceHandler (vuint8 status)`	
Multi Context	
`void ApplDescPostUserServiceHandler (vuint8 iContext, vuint8 status)`	
Parameter	
`iContext, status`	Refer 12.6.4.3 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 12.6.4.3 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 12.6.4.3 Service PostHandler for information. ■ **DescGetServiceId()** may be called here to dispatch the SID of the currently post-processed user service (refer 12.6.5.3 DescGetServiceId	

12.6.6 Session Handling

12.6.6.1 ApplDescCheckSessionTransition()

F I H

F 73553555

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
-

12.6.6.2 DescSessionTransitionChecked()

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	

12.6.6.3 DescIsSuppressPosResBitSet ()

Prototype	
Single Context	
`DescBool` **`DescIsSuppressPosResBitSet`** `(void)`	
Multi Context	
`DescBool` **`DescIsSuppressPosResBitSet`** `(vuint8 iContext)`	
Parameter	
`iContext`	the current request context location
Return code	
`kDescTrue` `kDescFalse`	The SPRMIB is set. 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	
■	

12.6.6.4 ApplDescOnTransitionSession()

F I

F 73553 5

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	

12.6.6.5 DescSetStateSession()

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 12.6.11.2 "DescSetState<StateGroup>()” for more details.	

12.6.6.6 DescGetStateSession()

I L

F 7 3 5 5 3 5

☒

☐

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 12.6.11.1 “DescGetState<StateGroup>()” for more details.

12.6.6.7 DescGetSessionIdOfSessionState

I L

F 3535

☒
☐

Prototype	
Any Context	
`DescMsgItem` **`DescGetSessionIdOfSessionState`** `(DescStateGroup sessionState)`	
Parameter	
`sessionState`	- Must be one of the valid session states (i.e. the value of the API *DescGetStateSession()*).
Return code	
`DescMsgItem`	- Is the corresponding session identifier value.
Functional Description	
This function provides a conversion from a session state to its corresponding session identifier (e.g. calling this function with parameter ***kDescStateSessionDefault*** will return ***0x01***).	
Pre-conditions	
-	
Call context	
-	
Particularities and Limitations	
■	

12.6.7 CommunicationControl Handling

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

12.6.7.1 ApplDescCheckCommCtrl()

F I H H H
F 7 3 5 5 3 5
☒

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	

12.6.7.2 DescCommCtrlChecked()

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	
■	

12.6.8 Periodic call of ‘Service MainHandler’

12.6.8.1 DescStartRepeatedServiceCall()

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()	

12.6.8.2 DescStartMemByAddrRepeatedCall()

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()	

12.6.9 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. In multiple PID mode, the application can decide in each PID main handler to use the ring buffer or not. However, if one of the PIDs has dynamic length, the ring buffer mechanism can not be used for any PID in the list.

Note
The ring buffer should only be used for long responses, because using the ring buffer instead of the linear buffer causes a runtime overhead.

12.6.9.1 DescRingBufferStart()

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 9.1.3 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 13.6).

12.6.9.2 DescRingBufferWrite()

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 first, 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

12.6.9.3 DescRingBufferCancel()

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

12.6.9.4 DescRingBufferGetFreeSpace()

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	
-	

12.6.9.5 DescRingBufferGetProgress()

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	
■	

12.6.10 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).

12.6.10.1 ApplDesc<Signal-Handler>()

F I A -M C
F 7 3 5 5 3 5
☒

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.	

12.6.10.2 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

12.6.11 State Handling (CANdesc only)

12.6.11.1 DescGetState<StateGroup>()

I L A L C
F 7 3 5 5 3 5
☒
☐

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
Funct Func S" nc	

12.6.11.2 DescSetState<StateGroup>()

I A L C
F 7 3 5 5 3 5

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

-From a task with priority lower or equal to the DescTask.

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

12.6.11.3 ApplDescOnTransition«StateGroup»()

F I L

F 7 3 5 5 3 5

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.	

12.6.12 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).

12.6.12.1 DescForceRcrRpResponse()

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().**	

12.6.12.2 ApplDescRcrRpConfirmation()

F I H

F 7 3 6 6 3 5

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).	

12.6.13 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.

12.6.13.1 DescMayCallStateTaskAgain()

I H F
F 9355

Prototype	
Single Context	
`DescBool` **`DescMayCallStateTaskAgain`** `(void)`	
Multi Context	
`DescBool` **`DescMayCallStateTaskAgain`** `(void)`	
Parameter	
–	-
Return code	
`kDescTrue` `kDescFalse`	TRUE if you may call again the state task within this application task cycle. 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

■

12.6.13.2 ApplDescCheckDynDidMemoryArea()

ApplDescCheckDynDidMemoryArea
F 3573355

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
•

12.6.13.3 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.

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

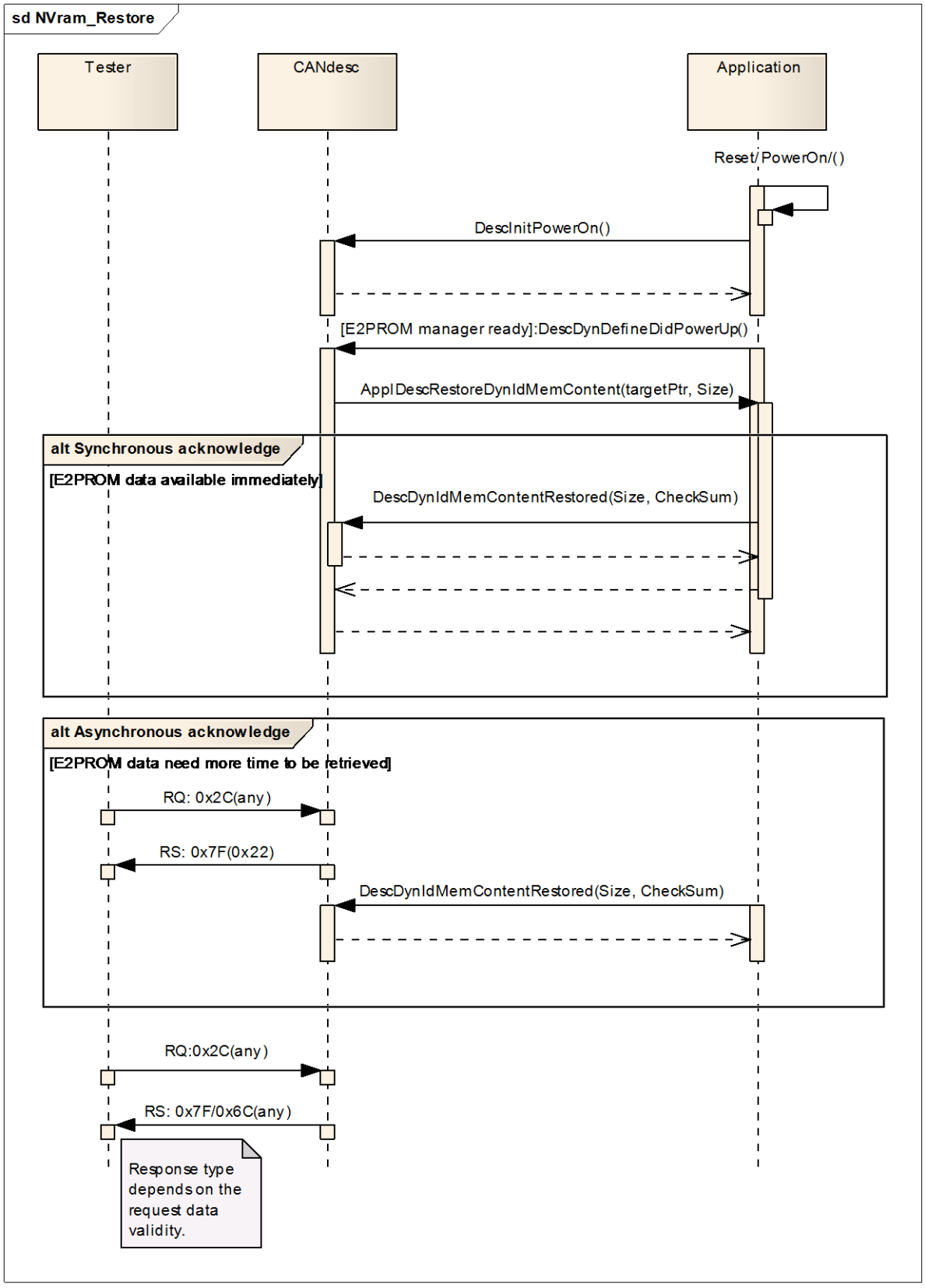


Figure 12-1 DynDID definition restore and tester interaction

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

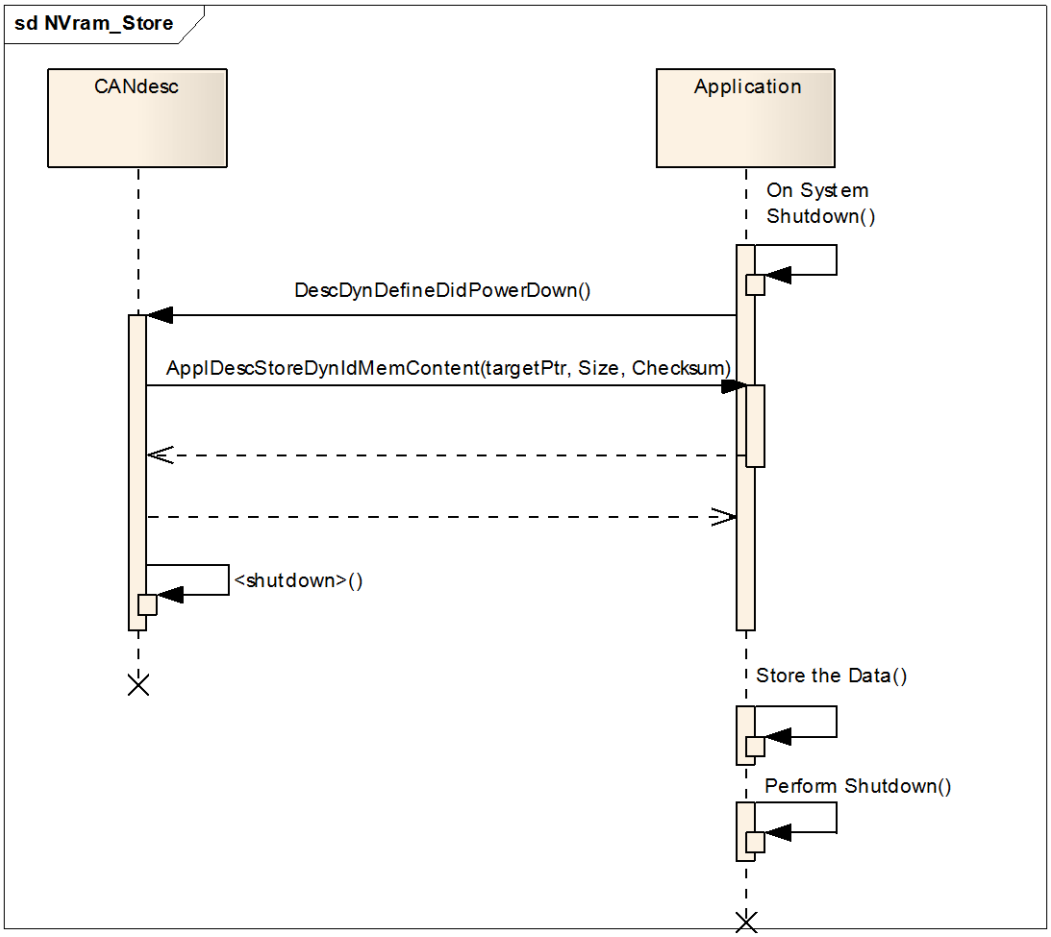


Figure 12-2 Store DynDID definitions

12.6.13.3.1 DescDynDefineDidPowerUp()

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().

12.6.13.3.2 DescDynIdMemContentRestored ()

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

12.6.13.3.3 DescDynDefineDidPowerDown ()

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.

12.6.13.3.4 ApplDescStoreDynIdMemContent ()

Prototype	
Single Context	
`void ApplDescStoreDynIdMemContent (DescDynDidStorageInfo storageInfo)`	
Multi Context	
`void ApplDescStoreDynIdMemContent (DescDynDidStorageInfo storageInfo)`	
Parameter	
`storageInfo.nvData`	The pointer to the data to be stored;
`storageInfo.nvDataSize`	The size (in bytes) of the table;
`storageInfo.checkSum`	The checksum value, calculated by CANdesc, to be stored.
Return code	

12.6.13.3.5 ApplDescRestoreDynIdMemContent ()

F I I H
F 3 3>

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

`storageInfo.nvData`
`storageInfo.nvDataSize`
`storageInfo.checkSum`
	The pointer to the data to where the stored data shall be written
	The size (in bytes) of the table expected.
	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	
■	

12.6.14 Memory Access Callbacks

12.6.14.1 ApplDescReadMemoryByAddress()

F I G F

F 3 3 9

☐
☒

Prototype	
Any Context	
`void ApplDescReadMemoryByAddress (DescMsgContext* pMsgContext, t_descMemByAddrInfo* pMemInfo)`	
Parameter	
`pMsgContext`	Refer the section 12.6.4.2 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()*).

- The read memory by address service is supported.
- Refer to chapter *9.3 Read/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.

From **DescTask()**

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

12.6.14.2 ApplDescWriteMemoryByAddress()

F I G F

F 3 3 9

☐
☒

Prototype	
Any Context	
`void ApplDescWriteMemoryByAddress (DescMsgContext* pMsgContext, t_descMemByAddrInfo* pMemInfo)`	
Parameter	
`pMsgContext`	Refer the section 12.6.4.2 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()*).

- The write memory by address service is supported.
- Refer to chapter *9.3 Read/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.

From **DescTask()**

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

12.6.15 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.

12.6.15.1 DescSendPosRespFBL()

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) ill be t.

12.6.15.2 ApplDescInitPosResFblBusInfo()

F I G
F 3<3 9
☐
☒

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. The FBL positive response feature is supported. From *DescSendPosRespFBL* context. ■ -	

12.6.16 Debug Interface / Assertion

12.6.16.1 ApplDescFatalError()

F I

F 7355355

☐
☒

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

At least on type of assertions are activated

Form ISR or task level. The interrupts might be disabled

- 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.

12.6.17 “Spontaneous Response” transmission

To implement the service $86 (Respone On Event) it is necessary to transmit a message without a previous request. If the same CAN ID have to be used for this reponse as for the diagnostics response, CANdesc provides an API to trigger the transmission.

12.6.17.1 DescApplSendSpontaneousResponse()

I F

F 3>35

Prototype	
Any Context	
`DescBool` **`DescApplSendSpontaneousResponse`**`(DescMsg resData, DescMsgLen resLen, t_descUsdtNetBus* pBusInfo)`	
Parameter	
`resData`	Pointer to an application buffer with response data (including posive response header).
`resLen`	Number of bytes to be sent (up to 4095 bytes).
`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	
`kDescTrue`	Operation was successful, the response will be sent.
`kDescFalse`	Operation failed – no response will be sent.
Functional Description	
Calling this function the application can force CANdesc to send immediately a spontaneous response. If CANdesc is currently busy with a tester request, there will be no response sent by this API and `kDescFalse` will be returned. If this API returns `kDescTrue`, the application shall wait for the *ApplDescSpontaneousResponseConfirmation()* prior initiating a new spontaneous transmission.	
Pre-conditions	
CANdesc was configured to use this option (enabled in the GENtool). Only possible to configure if Service 0x86 is contained in the cdd.	
Call context	
Task or interrupt.	
Particularities and Limitations	
■ -	

12.6.17.2 ApplDescSpontaneousResponseConfirmation()

F I H

F 3>35

Prototype	
Single Context	
`void ApplDescSpontaneousResponseConfirmation(vuint8 status)`	
Multi Context	
`void ApplDescSpontaneousResponseConfirmation (vuint8 iContext, vuint8 status)`	
Parameter	
`iContext`	Will be always “kDescPrimContext”.
`status`	If the transmission was successful, the parameter value will be *kDescOk*. Otherwise – *kDescFailed.*
Return code	
–	-
Functional Description	
Once the spontaneous response has been successfully triggered (ref. *DescApplSendSpontaneousResponse()*), this function will be called in any case. The transmission status is reported by the status parameter.	
Pre-conditions	
Only available if the API *DescApplSendSpontaneousResponse()* is available.	
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).	

12.6.18 Generic Processing Notifications

12.6.18.1 ApplDescManufacturerIndication

F I

F 3 3 5

Prototype
Single Context
`void ApplDescManufacturerIndication(vuint8 sid, vuint8* data, vuint16 length, vuint8 reqType, t_descUsdtNetBus* pBusInfo)`
Multi Context
`void ApplDescManufacturerIndication(vuint8 iContext, vuint8 sid, vuint8* data, vuint16 length, vuint8 reqType, t_descUsdtNetBus* pBusInfo)`

Parameter	
`iContext`	The current request context location (used only as a handle - *DO NOT MODIFY*).
`sid`	The service identifier of the received service request.
`data`	Pointer to the first byte of the request data (without service identifier byte).
`length`	Length of the request data (without service identifier byte)
`reqType`	The current request addressing method. Could be either ‚kDescFuncReq' or ‚kDescPhysReq' (bitmapped).
`pBusInfo`	The current request communication information (i.e. driver type (CAN, MOST, FlexRay, etc.), addressing information, communication channel number, tester address (if applicable) etc.

Return code	
–	-

Functional Description
This function is called right before CANdesc starts the processing of a received request.

Pre-conditions
Only available if the feature “Manufacturer Notification Support” is activated and CANdesc UDS2012 is used.
Call context
From **DescTask()**
Particularities and Limitations

12.6.18.2 ApplDescManufacturerConfirmation

F I H

F 36 35

☒

Prototype	
Single Context	
`void ApplDescManufacturerConfirmation(vuint8 status)`	
Multi Context	
`void ApplDescManufacturerConfirmation(vuint8 iContext, vuint8 status)`	
Parameter	
`iContext`	The current request context location (used only as a handle - *DO NOT MODIFY*).
`status`	**kDescPostHandlerStateOk** The positive response was transmitted successfully **kDescPostHandlerStateNegResSent** It was a negative response **kDescPostHandlerStateTxFailed** A transmission error occurred
Return code	
–	-
Functional Description	
This function is called after the processing of a request has been finished, a response has been sent (or sending has failed) and all service PostHandlers were called.	
Pre-conditions	
Only available if the feature “Manufacturer Notification Support” is activated and CANdesc UDS2012 is used.	
Call context	
From **DescTask ()**	
Particularities and Limitations	

12.6.18.3 ApplDescSupplierIndication

F I

F 3 3 5

Prototype
Single Context

`void ApplDescSupplierIndication(vuint8 sid,`
`vuint8* data,`
`vuint16 length,`
`vuint8 reqType,`
`t_descUsdtNetBus* pBusInfo)`

Multi Context

`void ApplDescSupplierIndication(vuint8 iContext,`
`vuint8 sid,`
`vuint8* data,`
`vuint16 length,`
`vuint8 reqType,`
`t_descUsdtNetBus* pBusInfo)`

Parameter	
iContext	The current request context location (used only as a handle - *DO NOT MODIFY*).
sid	The service identifier of the received service request.
data	Pointer to the first byte of the request data (without service identifier byte).
length	Length of the request data (without service identifier byte)
reqType	The current request addressing method. Could be either ‚kDescFuncReq' or ‚kDescPhysReq' (bitmapped).
pBusInfo	The current request communication information (i.e. driver type (CAN, MOST, FlexRay, etc.), addressing information, communication channel number, tester address (if applicable) etc.

Return code	
–	-

Functional Description
This function is called during the processing of a request, after CANdesc has verified that the requested service is allowed in the active session and security state.

Pre-conditions
Only available if the feature “Supplier Notification Support” is activated and CANdesc UDS2012 is used.
Call context
From **DescTask()**
Particularities and Limitations

12.6.18.4 ApplDescSupplierConfirmation

F I H

F 36 35

Prototype
Single Context
`void ApplDescSupplierConfirmation(vuint8 status)`
Multi Context
`void ApplDescSupplierConfirmation(vuint8 iContext, vuint8 status)`

Parameter	
`iContext`	The current request context location (used only as a handle - *DO NOT MODIFY*).
`status`	**kDescPostHandlerStateOk** The positive response was transmitted successfully **kDescPostHandlerStateNegResSent** It was a negative response **kDescPostHandlerStateTxFailed** A transmission error occurred
Return code	
–	-

Functional Description
This function is called after the processing of a request has been finished, a response has been sent (or sending has failed) and all service PostHandlers were called. It is called before *ApplDescManufacturerConfirmation()*.
Pre-conditions
Only available if the feature “Supplier Notification Support” is activated and CANdesc UDS2012 is used.
Call context
From **DescTask ()**
Particularities and Limitations

13 How To…

13.1 …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);
  Þescn01 242.6□606.67 Tm1 TmÛ gÛ G[( )] TÊTÊscn01 242. 663.46 Tm□0.024 Tc[(        )] T
}
```



```
kpMsgContext
```

```
                                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);
}
```

13.2 …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);
}
```

13.3 …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;
}
```

13.4 …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);
}
```

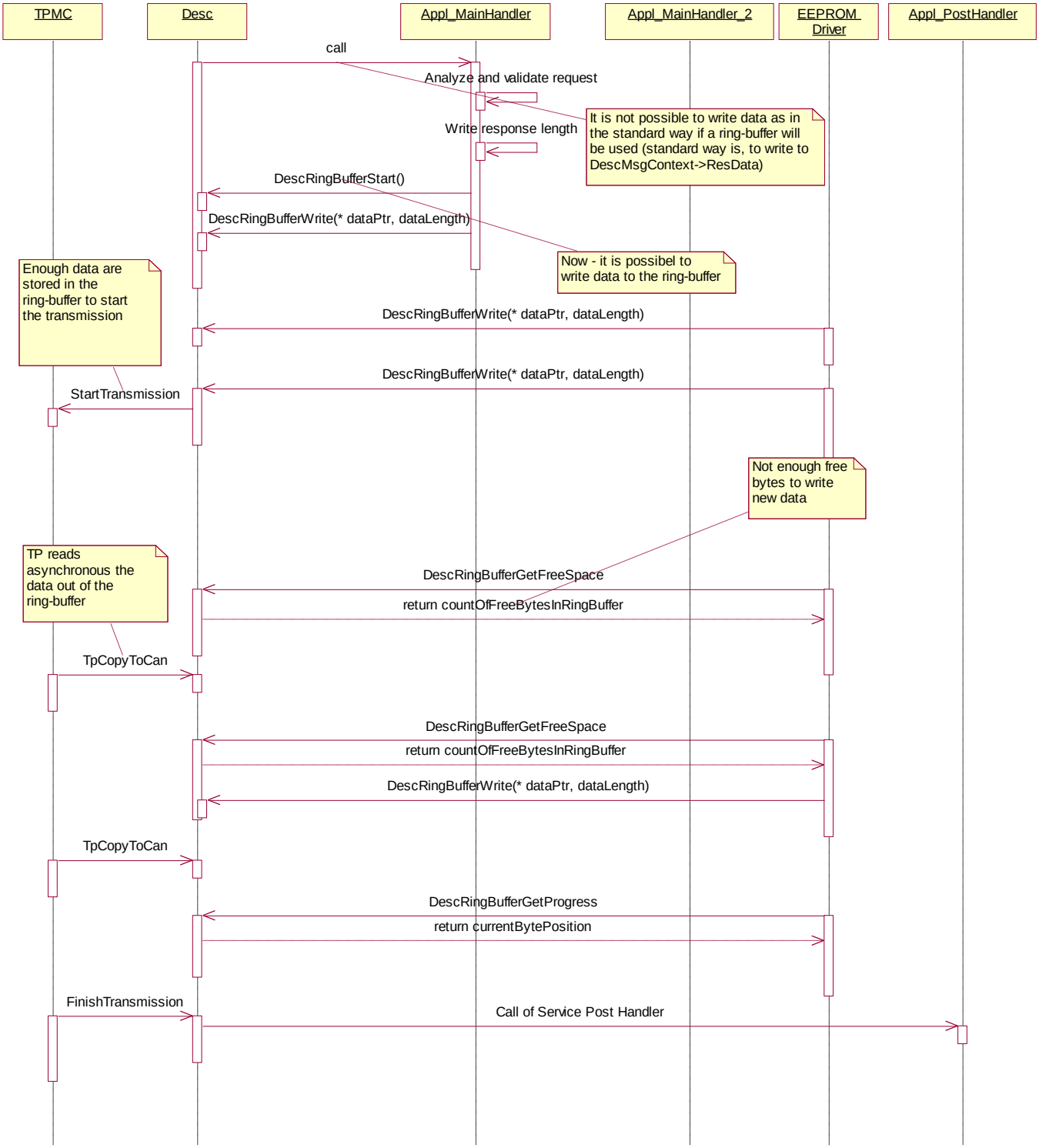
13.5 …implement a state transition function

```
//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". */
}
```

13.6 …work with the ring-buffer mechanism

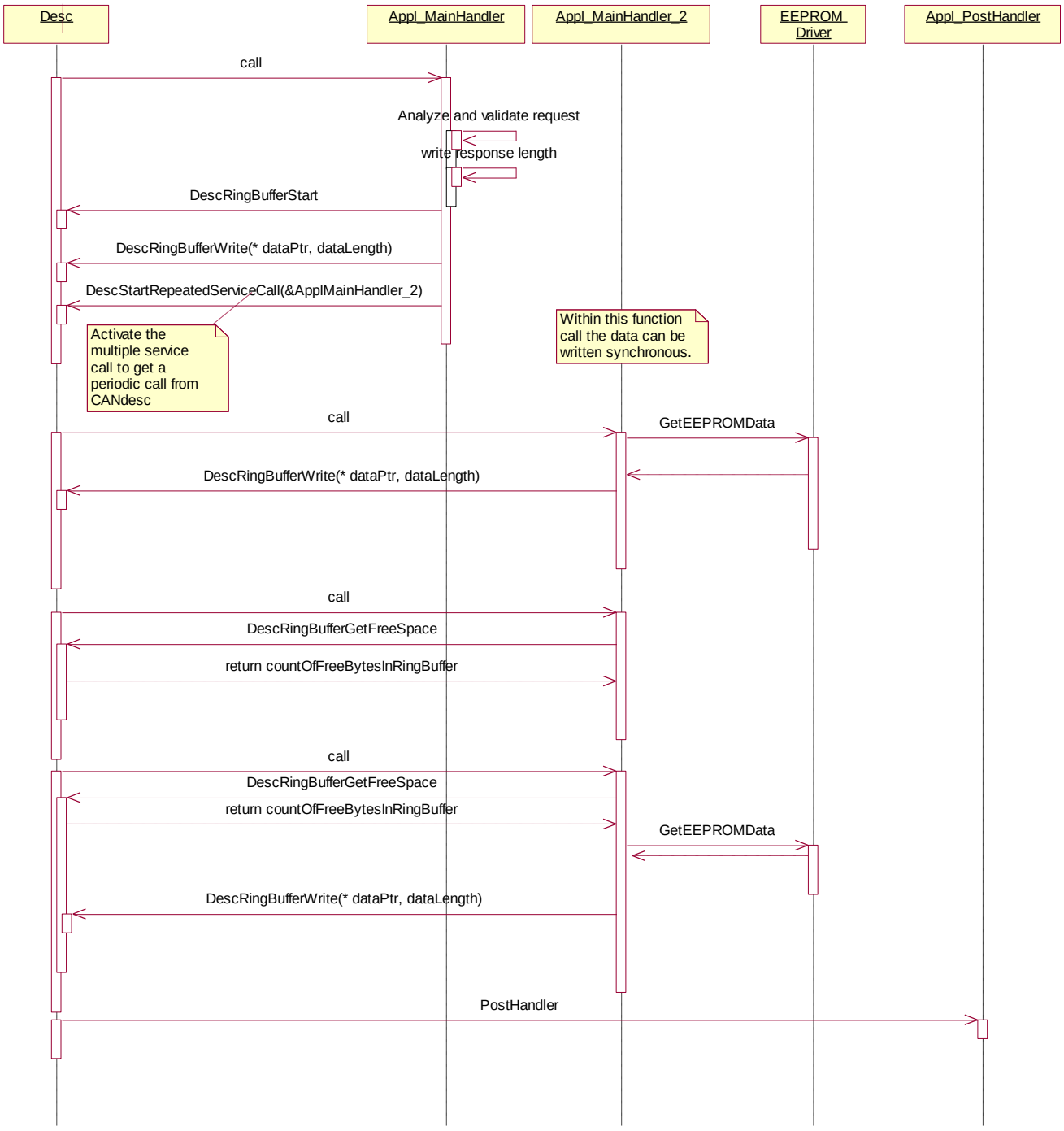
13.6.1 with asynchronous write




```
//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 */
  }
}
```

13.6.2 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 */
  }
}
```

13.7 …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 */
  }
}
```

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

According to some specifications the application has to send a positive response either to “diagnostic session control – default session” or “ECU reset – hard reset” after a successful flash job without a request. The Flash Boot Loader has to 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 (*ApplDescInitPosResFblBusInfo()*) to provide the concrete addressing information of the response.

4) Bus communication can be released by the application.

13.9 …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
```

13.10 …configure Extended Addressing

If Extended Addressing is used as TP Addressing mode some additional settings have to be done. “*DescCheckTA*” has to be set for the “*Target Address Message Filter*” in GENy.

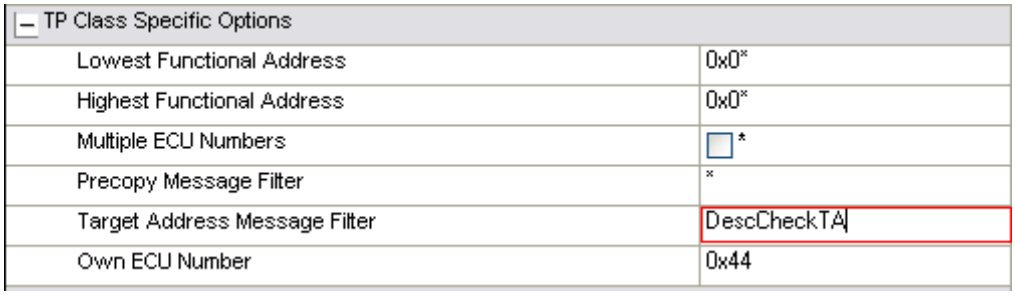


Figure 13-1 GENy TP configuration

Additional a user configuration file has to be used to configure the functional target address. An example for the content of the user configuration file is given below.

```
#define kDescOemExtAddrFuncTargetAddr  0xFE
```

13.11 …use Multiple Addressing

This chapter is a short summary of additional information that the application has to provide for CANdesc if the Tp addressing mode is Multiple Addressing.

In the case that a positive response has to be send after FBL flash job of the application, please assure that the correct addressing information are provided in the callback *ApplDescInitPosResFblBusInfo()*.

Furthermore, the “Rx Get Buffer” and “Rx Indication” functions have to be redirected to the application if one of the Tp Addressing modes is Normal Addressing. This can be done in the GENy configuration of the TP, a callback name different from the one that is implemented in CANdesc has to be entered.

Configurable Options	Diag
– TP Connection Group	
Number of Rx Channels	1
Number of Tx Channels	1*
Rx Get Buffer	DispatcherDescGetBuffer
Rx Indication	DispatcherDescPhysReqInd
Rx Error Indication	DescRxErrorIndication
Rx Single Frame Indication	*
Rx First Frame Indication	*
Rx Consecutive Frame Indication	*
Rx Copy from CAN	*
Rx Flow Control Frame Transmitted	*
Tx Confirmation	DescConfirmation
Tx Error Indication	DescTxErrorIndication
Tx Notification	*
Tx CAN Message transmitted	*
Tx Flow Control Frame received	*
Tx Copy to CAN	DescCopyToCAN
Tx Delay finished	*

Figure 13-2 GENy TP callbacks

The callbacks have to be implemented in the application. In the Get Buffer function the CAN Id has to be set for the FC in the case of Normal Addressing,

```
/* Example: A configuration with only CANdesc Tp connections and only one Tp
Tx/Rx channel. */
TP_MEMORY_MODEL_DATA canuint8* DispatcherDescGetBuffer(canuint8 tpChannel,
canuint16 datLen)
{
  TP_MEMORY_MODEL_DATA canuint8* retPtr = V_NULL;

  retPtr = DescGetBuffer(tpChannel, datLen);

   if(retPtr != V_NULL)
   {
    if((TpRxGetAddressingFormat(tpChannel) == kTpNormalAddressing))
    {
        /* kApplNormalAddressingTxId, have to defined by the application*/
        TpRxSetTransmitID(tpChannel, kApplNormalAddressingTxId);
    }
   }

 return retPtr;
}
```

The response ID for Normal Addressing has to be set in the Indication function. The response Id has to be set after the call of *DescPhysReqInd().*

```
/* Example: A configuration with only CANdesc Tp connections and only one Tp
Tx/Rx channel. */
void DispatcherDescPhysReqInd(canuint8 tpChannel, canuint16 datLen)
{
  vuint8 addressingType = (TpRxGetAddressingFormat(tpChannel));

   DescPhysReqInd(tpChannel, datLen);

   /*Set CAN IDs for the Response*/
   if(addressingType == kTpNormalAddressing)
   {
    /* kApplNormalAddressingTxId and kApplNormalAddressingPhysRxId, have to
    defined by     the    application*/
    TpTxSetChannelID(0 /*tpTxChannel*/, kApplNormalAddressingTxId,
    kApplNormalAddressingPhysRxId);
    /* tpTxChannel = 0 is only possible because only one Tx Channel is
    configured.*/
   }
}
```

13.12 …use “Dynamic Normal Addressing Multi TP” with multiple tester

This chapter is a short summary of additional information that the application has to provide for CANdesc if the Tp addressing mode is “Dynamic Normal Addressing Multi TP” with more than one tester.

In the case that a positive response has to be send after FBL flash job of the application, please assure that the correct addressing information are provided in the callback *ApplDescInitPosResFblBusInfo()*.

Furthermore, the “Rx Get Buffer” function has to be redirected to the application. This can be done in the GENy configuration of the TP, a callback name different from the one that is implemented in CANdesc has to be entered.

TP Connection Group	
Name	Diag
Number of Rx Channels	1*
Number of Tx Channels	1*
Rx Get Buffer	DispatcherDescGetBuffer
Rx Indication	DescPhysReqInd
Rx Error Indication	DescRxErrorIndication
Rx Single Frame Indication	*
Rx First Frame Indication	*
Rx Consecutive Frame Indication	*
Rx Copy from CAN	*
Rx Flow Control Frame Transmitted	*
Tx Confirmation	DescConfirmation
Tx Error Indication	DescTxErrorIndication
Tx Notification	*
Tx CAN Message transmitted	*
Tx Flow Control Frame received	*
Tx Copy to CAN	DescCopyToCAN

Figure 13-3 GENy TP callbacks (physical addressing)

The “Get Buffer” function of the functional connection has to be redirected to the application too.

TP Functional Connection Group	
Get Buffer (mandatory)	DispatcherDescGetFuncBuffer
Indication (mandatory)	DescFuncReqInd
Copy from CAN	*

Figure 13-4 GENy TP callbacks (functional addressing)

The callbacks have to be implemented in the application. The received CAN ID has to be mapped to the corresponding transmit CAN ID and the TP connection number has to be set in the xxxGetBuffer callback:

```
TP_MEMORY_MODEL_DATA canuint8* DispatcherDescGetBuffer(canuint8 tpChannel,
canuint16 datLen)
{
  TP_MEMORY_MODEL_DATA canuint8* retPtr = V_NULL;

  switch(TpRxGetChannelID(tpChannel))
  {
  case kDispatcherRxDiagPhysCanId:
    TpRxSetTransmitID(tpChannel, kDispatcherTxDiagPhysCanId);
    TpRxSetConnectionNumber(tpChannel, kDescDiagConnection);
    retPtr = DescGetBuffer(tpChannel, datLen);
    break;

  case kDispatcherRxDiagAddPhysCanId:
    TpRxSetTransmitID(tpChannel, kDispatcherTxDiagAddPhysCanId);
    TpRxSetConnectionNumber(tpChannel, kDescDiagAddConnection);
    retPtr = DescGetBuffer(tpChannel, datLen);
    break;

  default:
    ;
  }
  return retPtr;
}
```

The receiced CAN ID has to be mapped to the corresponding transmit CAN ID in the xxxGetFuncBuffer callback. Furthermore it is important, that the physical Rx ID is set for the response and not the functional one. This CAN ID is used to recognize the FC of the tester in the used n]es]"

The code examples above are for 2 testers, in the example are some defines used that have to be provided by the application corresponding to the configuration.

Define	Description
kDispatcherRxDiagPhysCanId	Physical request CAN ID of the first tester
kDispatcherRxDiagFuncCanId	Functional request CAN ID of the first tester
kDispatcherTxDiagPhysCanId	Response CAN ID of the first tester
kDispatcherRxDiagAddPhysCanId	Physical request CAN ID of the second tester
kDispatcherRxDiagAddFuncCanId	Functional request CAN ID of the second tester
kDispatcherTxDiagAddPhysCanId	Response CAN ID of the second tester
kDispatcherTxDiagTpChannel	Transmit Tp Channel of CANdesc. If only one Tp Channel is used, it is has to be set to zero.

14 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.

15 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
VSG	**V**ehicle **S**ystem **G**roup

16 Contact

Visit our website for more information on

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

www.vector.com