User Manual

CANdesc

A Step by Step Introduction

Version 1.7

English

Impressum

Vector Informatik GmbH
Ingersheimer Straße 24
D-70499 Stuttgart

Manual History

Author	Date	Version	Details
Klaus Emmert	2004-05-10	1.1	Vector symbols included, template version 1.8 used (this history included), AppDesc… changed to ApplDesc due to software modifications, description of GENy as generation tool added, testing of diagnostics layer described with CANoe demo configuration, further Information about diagnostic buffer (linear and ring buffer mechanism) and the repeated service call feature
Klaus Emmert	2004-10-15	1.2	Modifications after Review.
Klaus Emmert	2005-08-12	1.3	Two new functions: DescTimerTask(), DescStateTask(). These two functions can be used instead of DescTask to handle the timers and the application separately.
Klaus Emmert	2006-03-24	1.4	Issues in example code fixed Document overview added
Oliver Garnatz	2007-01-12	1.5	Added description of CANdesc_ConnectorCAN GENy component
Klaus Emmert	2008-01-28	1.6	References fixed
Manuela Scheufele	2009-07-27	1.7	(see section Version 1.7 on page 66)

Reference Documents

No.	Source	Title
[1]	Vector Informatik	Technical Reference CANdesc
[2]	Vector Informatik	Technical Reference CANdescBasic

Inhaltsverzeichnis

1 Manual Information

In this chapter you find the following information:

1.1 About this user manual

Finding information quickly

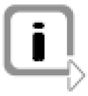
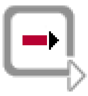
The user manual provides the following access help:

- At the beginning of each chapter you will find a summary of the contents,
- In the header you can see in which chapter and paragraph you are,
- In the footer you can see to which version the user manual replies,
- At the end of the user manual you will find an index, with whose help you will quickly find information,
- Also at the end of the user manual you will find a glossary in which you can look up an explanation of used technical terms

Conventions

In the two following charts you will find the conventions used in the user manual regarding utilized spellings and symbols.

Style	Utilization
bold	Blocks, surface elements, window- and dialog names of the software. Accentuation of warnings and advices. **[OK]** Push buttons in brackets **File|Save** Notation for menus and menu entries
MICROSAR	Legally protected proper names and side notes.
`Source Code`	File name and source code.
Hyperlink	Hyperlinks and references.
<CTRL>+<S>	Notation for shortcuts.

Symbol	Utilization
	Here you can obtain supplemental information.
	This symbol calls your attention to warnings.
	Here you can find additional information.
	Here is an example that has been prepared for you.
	Step-by-step instructions provide assistance at these points.
	Instructions on editing files are found at these points.
	This symbol warns you not to edit the specified file.

1.1.1 Certification

Certified Quality Management System	Vector Informatik GmbH has ISO 9001:2000 certification. The ISO standard is a globally recognized standard.
Spice Level 3	The Embedded Software Components business area at Vector Informatik GmbH achieved process maturity level 3 during a HIS-conformant assessment.

1.1.2 Warranty

Restriction of warranty	We reserve the right to change the contents of the documentation and the software without notice. Vector Informatik GmbH assumes no liability for correct contents or damages which are resulted from the usage of the documentation. We are grateful for references to mistakes or for suggestions for improvement to be able to offer you even more efficient products in the future.

1.1.3 Registered trademarks

Registered trademarks	All trademarks mentioned in this documentation and if necessary third party registered are absolutely subject to the conditions of each valid label right and the rights of particular registered proprietor. All trademarks, trade names or company names are or can be trademarks or registered trademarks of their particular proprietors. All rights which are not expressly allowed are reserved. If an explicit label of trademarks, which are used in this documentation, fails, should not mean that a name is free of third party rights.

→ Outlook, Windows, Windows XP, Windows 2000, Windows NT, Visual Studio are trademarks of the Microsoft Corporation.

1.1.4 Errata Sheet of manufacturers

Caution: Vector only delivers software!

Your hardware manufacturer will provide you with the necessary errata sheets concerning your used hardware. In case of errata dealing with CAN please provide us the relevant erratas and we will figure out whether this hardware problem is already known to us or whether to get a possible workaround.

Info: Because of many NDAs with different hardware manufacturers or because we are not informed about, we are not able to provide you with information concerning hardware errata of the hardware manufacturers.

2 Getting Started

In this chapter you find the following information:

2.1 How to use this Manual

Just follow the description step by step.

FAQ

To find answers to special questions without reading the whole document use the FAQ list (see section FAQs on page 63).

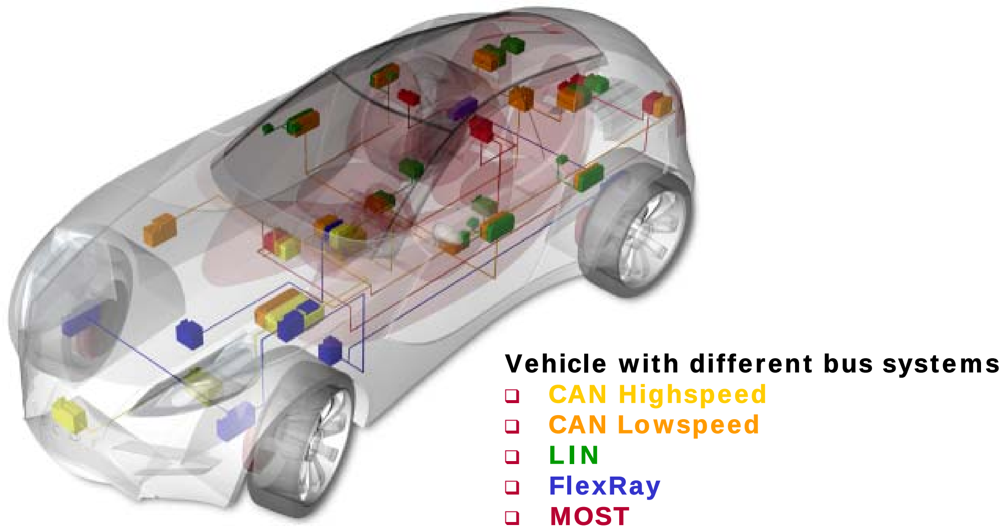
3 Basic Information

In this chapter you find the following information:

3.1 An Overall View

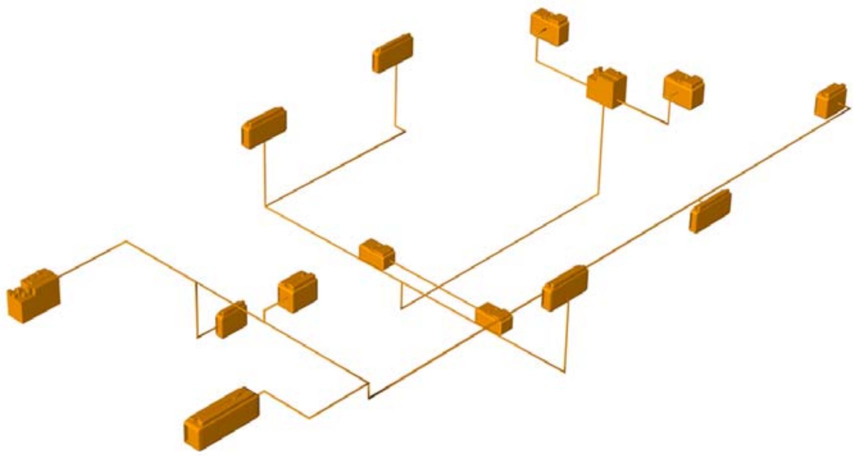
ECU in the focus

What we are now talking about is an ECU, a module to be built-in a vehicle like shown in the figure below. Almost every ECU participates in a certain bus system like e.g. CAN, FlexRay or LIN.



So any ECU within one bus system has to provide an identical interface to this bus system because all ECUs have to share information via this bus system as you see in the figure below.

CAN Lowspeed as an example bus system



For that reason all ECUs are built-up in the same way. There is a software part to realize the main job (application) of this ECU e.g. to control the engine or a door. The other part is the software part to be able to communicate with the other ECUs via the bus system that is the communication software.

3.2 What is Diagnostic

Dia'gno stics - Detection, Examination of a machine;

[greek. *diagnoskein* „analyze deeply, differentiate]

In contrast to Dia’gno · sis – Examination (med.)

Diagnostics in a technical context is the examination of a machine. But diagnostics in this context goes way beyond this definition.

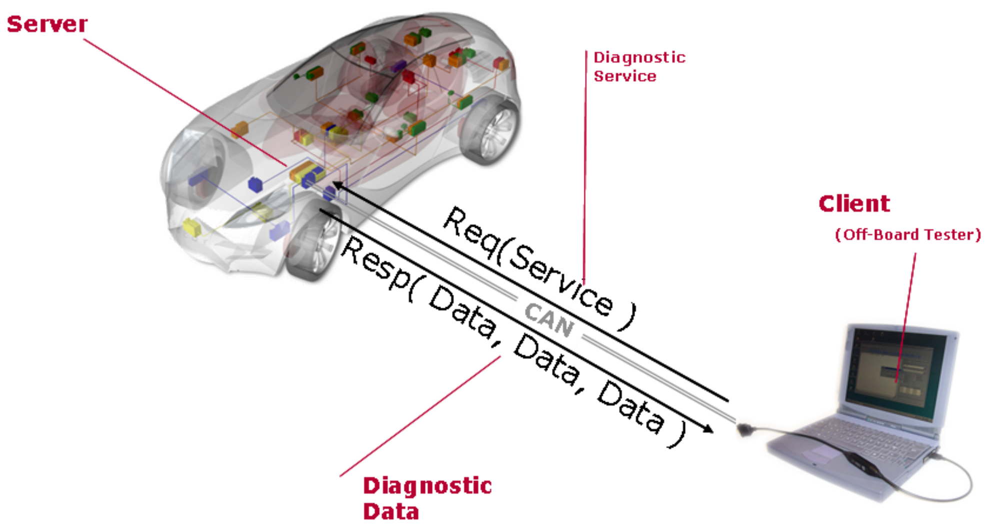
Diagnostics comprises function monitoring, error detection, fault memory, activation, data acquisition etc. and is used for variant coding, end-of-line programming, reprogramming, identification etc.

3.3 What happens during Diagnostics?

In most cases an Off-Board tester (Client) sends a diagnostic request to the ECU (via CAN) and the ECU (Server) sends back a diagnostic response. This can be a positive or a negative response. The following figure clearly shows a basic representation of this mechanism.

CANdesc –

CAN Diagnostic Embedded Software Component



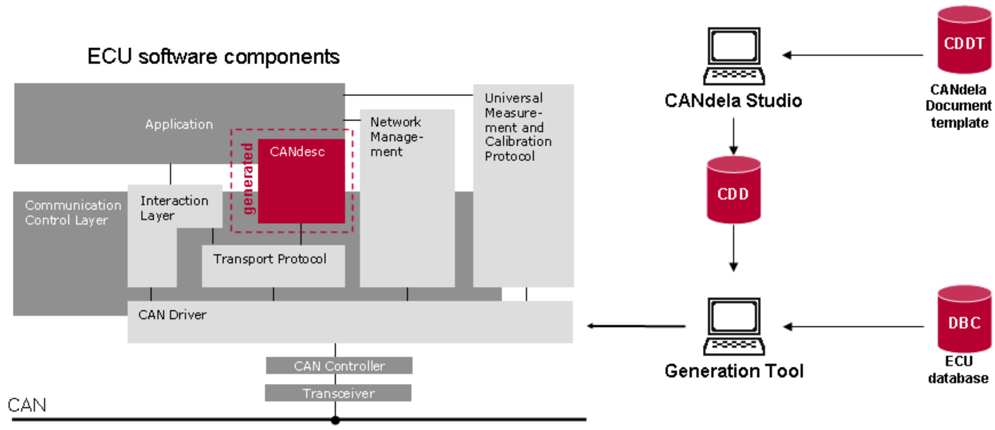
3.4 What is CANdesc?

CANdesc is totally generated based upon the CDD file.

CANdesc stands for **CAN Diagnostic Embedded Software Component**.

This software component differs from all other CANbedded Software Components in that it is totally generated. To be able to generate this component you need a CDD file, a DBC file and the generation tool (GENy / CANgen).

Generated Software Component based on .CDD and .DBC



Info: The CANdesc will be explained in the section Generic Handling of a Diagnostic Request in the CANdesc Component on page 21, where you will get detailed insight into the CANdesc Component and how it works when processing a diagnostic request.

3.5 Tools and Files

3.5.1 CANdela Studio, CDDT, CDD

All settings you have to do in CANdela Studio to use CANdesc are stored in the CDD file.

CANdela Studio is a PC tool. It reads in the diagnostic template file CDDT and generates a diagnostic data base, the CDD file.

The CDDT is a description of the OEM diagnostic specification.

All necessary diagnostic information, such as supported diagnostic services, sub services, format, signals, state filters, state transitions etc., is described via CANdela Studio and stored in the CDD file.

To use the CANdesc component, you need the CDD file and you need to know how to make the necessary settings in CANdela Studio.

3.5.2 Generation Tool, CDD, DBC

Remember to add the path to the CDD file in the Generation Tool

The generation tool (GENy / CANgen) is a PC Tool, too. It generates configuration files and signal interface files for the CANbedded Software Components. The generation tool needs the DBC file to generate the files.

There is the same DBC file per bus

The DBC file is designed by the vehicle manufacturer and distributed to all suppliers that develop an ECU. Thus every supplier uses the SAME DBC file for one vehicle platform and one bus system (powertrain, body CAN etc.) to guarantee a common system (high speed, low speed, etc) for all suppliers to guarantee a common basis for development

basis for development.
For example, every ECU has to know that a 1 in bit 7 in the 4th byte of the message 0x305 means “Ignition Key” on/off.

The DBC file contains information about every node and the messages / signals the node has to send and to receive.

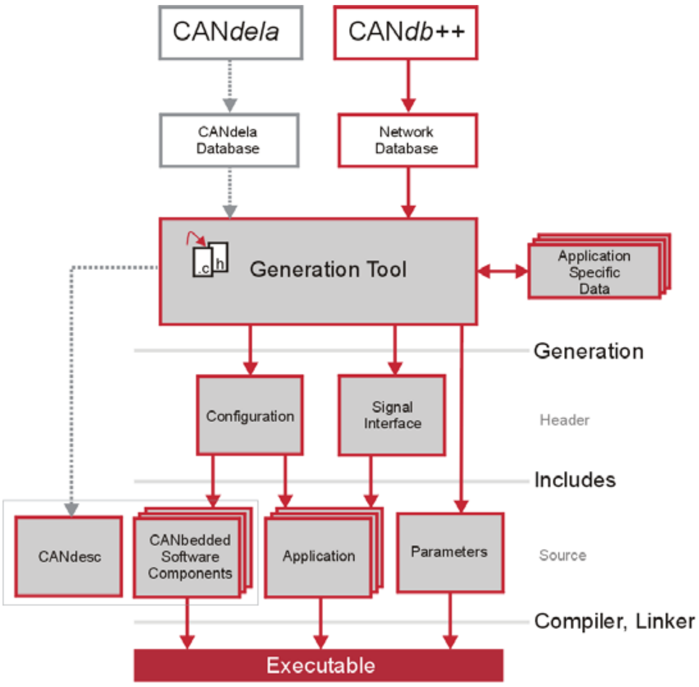
When using CANdesc for diagnostics the CDD file must be read in by the generation tool, to be able to generate the CANdesc code.

3.5.3 Generation Process with CANbedded Software Components

Normally the generation tool generates files that contain the configuration and the signal interface of the CANbedded Software Components. CANbedded can be compiled and linked using the source code of each component.

The standard generation process for Vector Software Components.

CANdesc is a completely generated Software Component



The main difference for CANdesc is that the source code for CANdesc is totally generated from the CDD file and therefore not included in your delivery as the other software components are. Since the CDD file contains most of the information about CANdesc, there are only a few configuration settings left that can be done via the generation tool on the CANdesc tab

3.6 What CANdesc does

Handles Diagnostic Communication

- → CANdesc receives addressed requests physically or/and functionally
- → CANdesc generates and handles a physical or functional request with appropriate response message headers, corresponding to the given KWP2000/UDS (ISO 14229-1) Diagnostics on CAN manufacturer specification.
- → CANdesc connects to underlying Transport Protocol and handles the communication errors of the underlying layers.

- CANdesc is capable of communication on any bus systems, using an own abstraction interface.

Manages Diagnostic Data (Buffer)

- CANdesc keeps the data consistency, which guarantees that no other request will delete the current diagnostic request data being processed.

Handles Diagnostic Errors

- CANdesc prrovides centralized diagnostic error handling based on the method **report only first detected error**.
- CANdesc monitors timeouts (e.g. S3- “Tester Present”, P2- “Response pending”, etc.).

Analyzes Requests (state machine, filtering)

- CANdesc detects relevant SID (Service Identifier) for the ECU. If an SID is not supported by the current configuration, the appropriate reaction will be executed (e.g. negative response or the request will be ignored).
- CANdesc analyzes the service instance. This includes recognition of the service-specific sub functions for each supported SID. The request length is validated if it is defined to be constant. For dynamic fields, the application must do range checking of the request length.
- CANdesc validates the states. The component ensures that a service is only executed if the diagnostic state allows the processing of that service. E. g. some services are only allowed to be executed inside a special diagnostic session. If the current state does not allow the execution, a corresponding negative response is sent automatically.

Processes the request (optional)

- CANdesc generates a complete diagnostic handler function which fills out the correct response data for the application.
- CANdesc generates signal handlers to help the application place the response information.
- CANdesc generates a Service MainHandler which will use data access functions provided by the application, but will place the information on the message as defined in the diagnostic data description.
- CANdesc dispatches incoming request(s) to the application (Service MainHandler or signal handler level).

3.7 Diagnostics – a more detailed View

In this chapter you find the following information:

3.7.1 Basic Nomenclature from the Bottom Up

Using the same expressions does not mean to talk about the same thing

Basic diagnostic communication is based upon a request / response mechanism. To understand the structure of CANdela Studio it is necessary to make some detailed naming definitions.

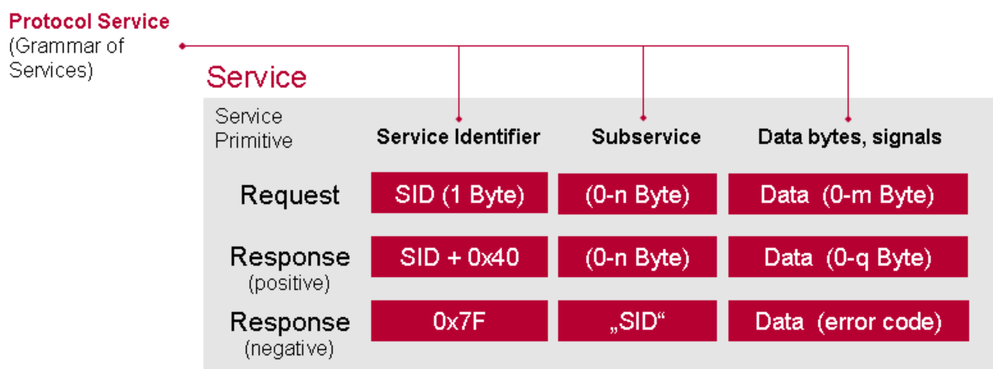
The combination of a request and responses (positive and negative) forms a **Service**, as you can see in the figure below. A service (in the scope of CANdesc) is a concrete service of an ECU.

This nomenclature should help to proceed with CANdesc and CANdela.

Request and responses are so-called service primitives.

Service Identifier = SID

Build-up of Requests and Response Messages



Service

Protocol Service

Request

Response

Diagnostic Instance

Diagnostic Class

A **protocol service** is a pattern for a service. The protocol service defines how the service primitives have to be built up. It determines the number and meaning of bytes for the sub service, and specifies the data bytes.

Info: The order of service identifier, sub service and data bytes can be found at the byte stream level, too.

Request

A request is a service primitive and is created as shown in figure above. A request is always sent from a tester to an ECU. The ECU processes the request and has to send back a response message.

Response

The positive response is calculated very easily by just adding the value 0x40 (hex format) to the SID of the request. The sub service is just repeated from the request and the data depends on the service.

The negative response always starts with 0x7F as the SID followed by the SID of the request. The error code shows the reason for the negative response (e.g. wrong format of the request, …).

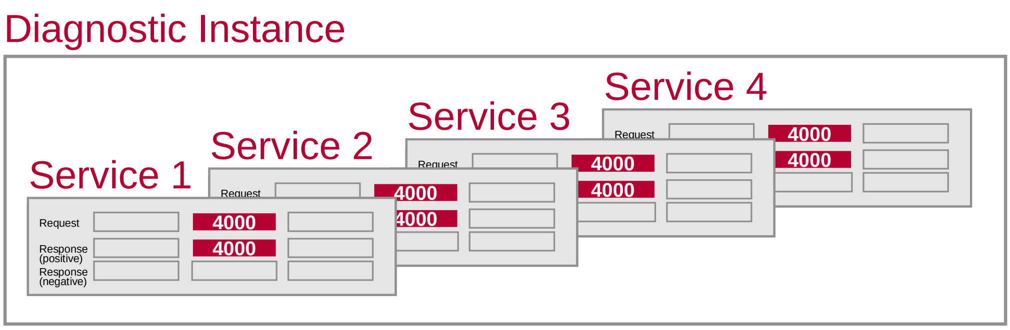
Services with the same sub service (similar functional scope) are combined into the same **Diagnostic Instance**. This sub service is the characteristic factor for the diagnostic instance.

A diagnostic instance is a part of a **diagnostic class**.

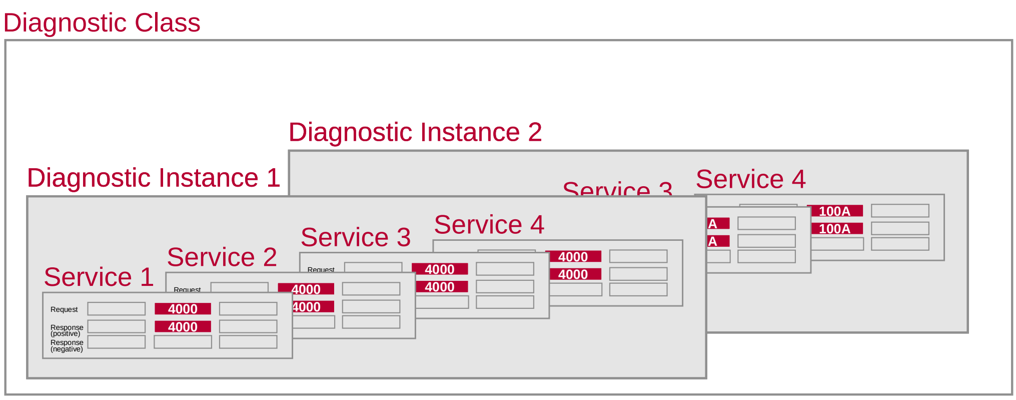
A diagnostic class is the abstract description of a use case.

This is shown in the following two illustrations.

Services with the same Subservice are combined into a Diagnostic Instance - the Sub Function 4000 is Just an Example



A Diagnostic Instance is a part of a Diagnostic Class



3.7.2 The same Nomenclature from the Top Down

CANdela is top down, CANdesc bottom up – try to understand both directions.

A **diagnostic class** is an abstract description of a use case.

A **diagnostic instance** is derived from a diagnostic class. Some diagnostic classes can be instantiated only once. Any diagnostic instance is unique and can be distinguished from another diagnostic instance via its sub service (e.g. data identifier). A diagnostic instance contains services.

Services are composed of the three **service primitives**: request, positive response and negative response. The **protocol service** is the pattern for the service, the grammar definition.

The service primitive **data** is a concrete information unit exchanged between the tester and the ECU. In the automotive environment you call them signals, too.

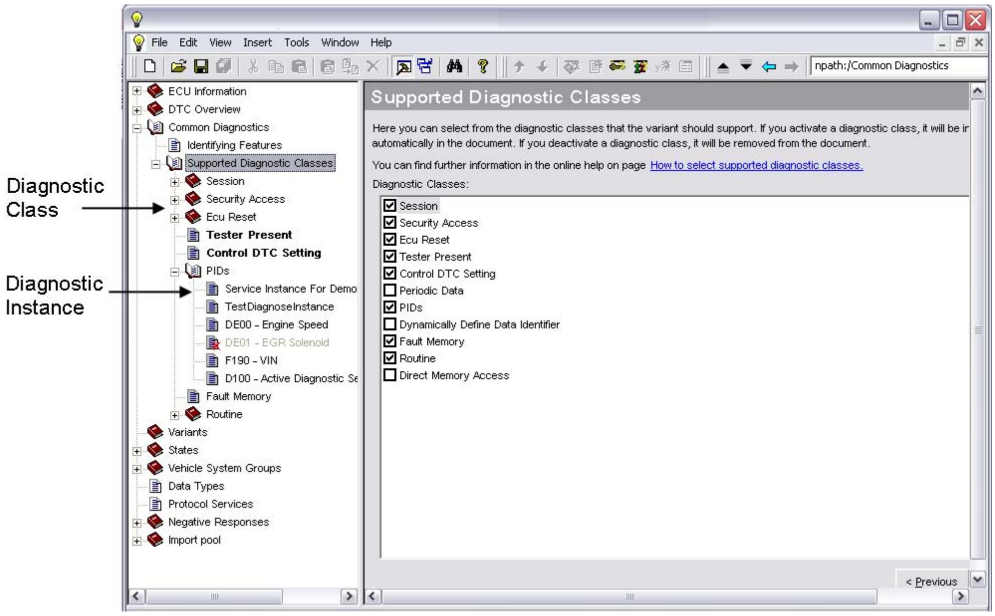
3.7.3 Where to find this Nomenclature in CANdela Studio

Getting around in CANdela Studio

To generate CANdesc you will have to make settings in the CDD file, i.e. you will have to work with CANdela Studio. That's the reason why it is very important that you get to know the areas in the CANdela Studio where to make the necessary settings.

Below there is a screenshot of CANdela Studio.

See the Diagnostics Classes and Diagnostic Instances in the CANdela Studio tree structure

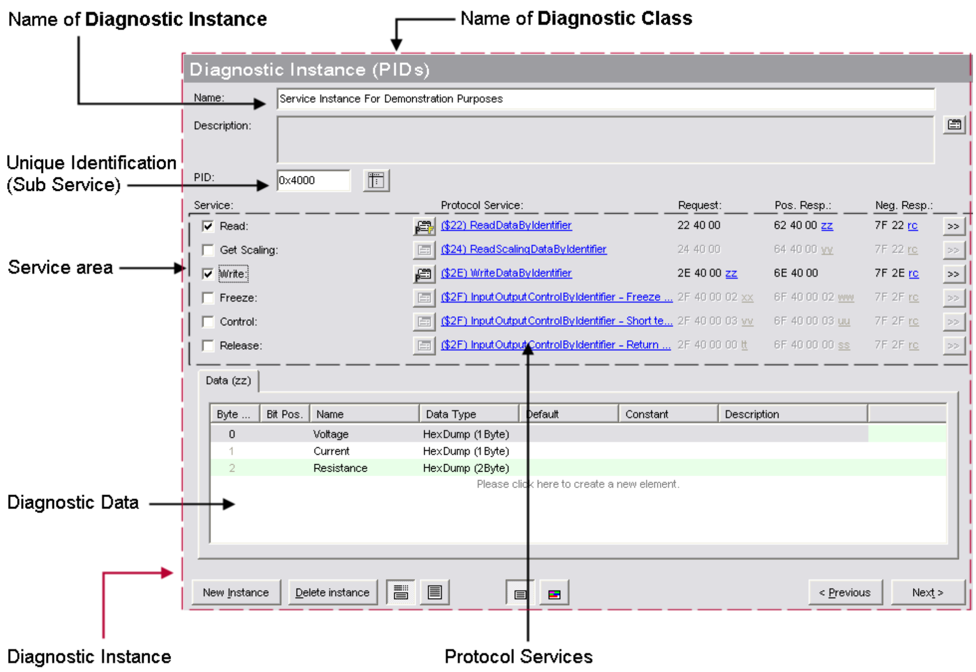


The structure within CANdela Studio is top down. In the tree on the left of CANdela Studio you will find the diagnostic class and the diagnostic instances as shown in the figure above.

Info: To get familiar with the idea of diagnostic classes and diagnostic instances, have a look at all supported diagnostic classes. Verify for yourself what is meant by **abstract description of a use case**, e.g. talking about Sessions, Security Access, Fault Memory…

If you click on a Service Instance you get a window like the following figure. Use this figure to understand the different areas on the diagnostic instance window and to close the gap between the nomenclature in the section above and it appears in CANdela Studio.

Diagnostic Instance window of CANdela Studio – a very important window



3.7.4 Generic Handling of a Diagnostic Request in the CANdesc Component

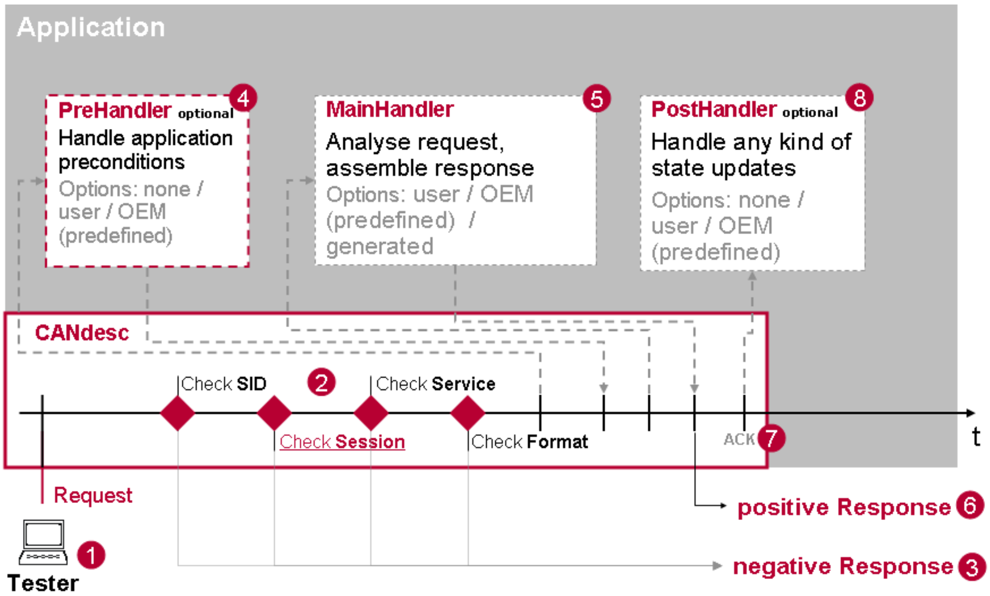
What happens in the CANdesc if a diagnostic message received?

Now you know the basic diagnostic elements and the build-up of diagnostic services. Now we take a closer look at how the diagnostic services are processed by CANdesc. You also need to know these processing steps so you can control and adapt this process.

Info: For this adaptation you have to use CANdela Studio.

The following figure shows the processing of a diagnostic service in detail.

Processing a Diagnostic Message received by CANdesc and the connections to the Application.



❶ Everything starts with a diagnostic request from a tester to the ECU.

Info: The path of this message through the CAN Driver and the Transport Protocol is not shown in the illustration.

❷ Now this incoming diagnostic request will be checked in different ways. Is the SID supported in the ECU? Is this SID supported in the current session? Is the service supported? Is the format of this request message correct, i.e. correct length? Correct data? etc.

❸ If any of these checks fail a negative response is sent back to the tester. The error code informs about the reason (e.g. wrong format).

❹ If the incoming diagnostic request passes all of these checks, a PreHandler function could be called. This PreHandler function is optional. You have options to set it to <none>, <user> or <OEM>.

❺ The next function is the MainHandler. This is a mandatory function. Every service must provide a MainHandler. The MainHandler is designed to analyze the request and assemble the response message. The MainHandler provides the options <user>, <OEM> and <generated>.

❻ After the MainHandler has processed the diagnostic data, provided the data for the response and informed the CANdesc Component about the end of the processing (processing done), the positive response message will be sent back to the tester.

Info: The path through the Transport Protocol and the CAN Driver is not shown in the figure above.

❼ After the diagnostic response is sent by the transport layer (ACK)**…**

❽ **…**the call of the PostHandler function is triggered. This function is optional too and can be set to <none>, <user> and <OEM>. Use this function to do any kind of state updates.

Info: A typical example for the **PostHandler** is to reset the CPU to start the bootloader.

3.7.5 User, None, OEM, Generated – what does this mean?

As you have read in the section above, a Pre-, Main- and PostHandler can be selected for any service to process the diagnostic service in a very user-friendly manner.

All handlers can be defined via CANdela Studio

Handler	Selectable settings
PreHandler	none, user, OEM
MainHandler	user, OEM, generated
PostHandler	none, user, OEM

None

None can be selected for Pre and PostHandlers only because these handlers are optional. As the name says, none switches the handler off.

User

The setting user means that you have to do the complete code for this handler. The function prototype is generated in appdesc.h.

OEM (predefined)

The setting OEM handles the request as required by the car manufacturer. The implementation is part of the CANbedded Software Component. The user does not have to add anything.

Info: The setting OEM should only be used if it is predefined.

Generated (Signal Handler)

If you select **Generated** you have two options for this handler (MainHandler)

1. Generate a function prototype (appdesc.h). Use this function to handle the diagnostic data by returning the current value (reading service) or using the parameter (writing service).
2. In CANdela Studio you can enter the name of the variable. In appdesc.h the external declaration of this variable is generated and you only need to define this variable in your application and that's all. Your application now just has to keep the content up to date.

Cross reference: For more details about the using the handlers and how to make the settings in CANdela refer to STEP Functional Connection between your Application and CANdesc/CANdela Studio on page 35.

4 A Few STEPS to CANdesc

In this chapter you find the following information:

4.1 STEP What do you need before start?

Check before you start

Before you start make sure that you have received everything you need.

CANbedded

Did you get the CANbedded delivery?

YES? Then go on

Except for the converter, you should answer all other questions with **yes** before going on here.

4.2 Startup Code

It is your responsibility

The **startup code** of the microcontroller is not part of the Vector delivery. The **startup code** complete is in your responsibility.

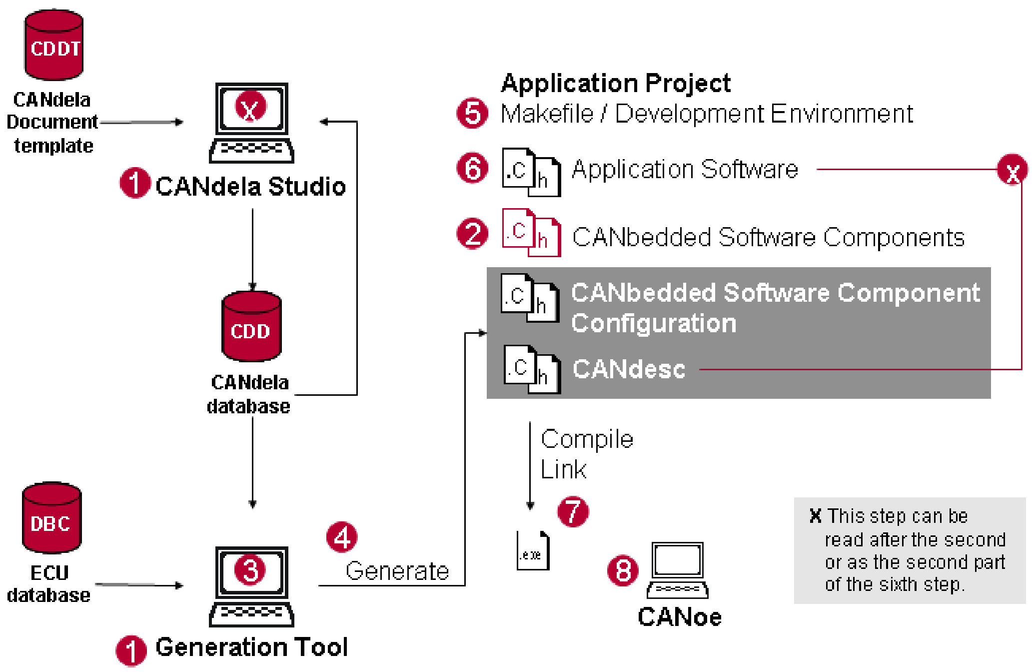
Take care to provide an appropriate **startup code** regarding e.g. wait states, etc.

Info: The **startup code** is not part of the Vector delivery.

4.3 Overview

Step overview

This overview shows the steps to CANdesc. These steps are described in detail on the following sections.



4.4 STEP Installation

As you see in the picture before, you need 2 PC tools to work with CANbedded containing CANdesc as a diagnostic component.

Generation Tool

The first tool is the generation tool. It is delivered with the CANbedded Software Components. Extract the files to an appropriate folder and follow the installation instructions.

Info: There are two kinds of generation tools, CANgen and GENy. Which of them you have to use depends on the delivery. In the following steps the usage of both tools are shown.

CANdela Studio

The second PC tool is CANdela Studio. This tool is for editing the *.CDD file. Install the tool by following the installation instructions.

Extract CANbedded Software Components

The number of CANbedded components in your delivery depends on your project.

To use CANdesc you need at least a **CAN Driver** and a **Transport Protocol** (e.g. OSEK / ISO 15765-2).

Copy all C and H files which are necessary for the components into your application project folder.

Cross reference: Refer to the corresponding user manuals (e.g. CANDriver User Manual) to get further information about the files of the different Software Components.

Info: Since CANdesc is totally generated, you won’t find any source files for CANdesc in your delivery.

4.5 STEP Configuration with the Generation Tool

As described above there are two generation tools for configuring the CANbedded Software Components, CANgen and GENy.

In the following chapters we describe the handling of both tools, beginning with CANgen. Figure out which tool you use and read the corresponding chapters only.

4.5.1 Using the Generation Tool CANgen

Open CANgen. Add a data base (DBC file) via the green plus .

Info: Normally you get a data base (DBC) from your vehicle manufacturer that is designed for your project.

Are the files generated in the

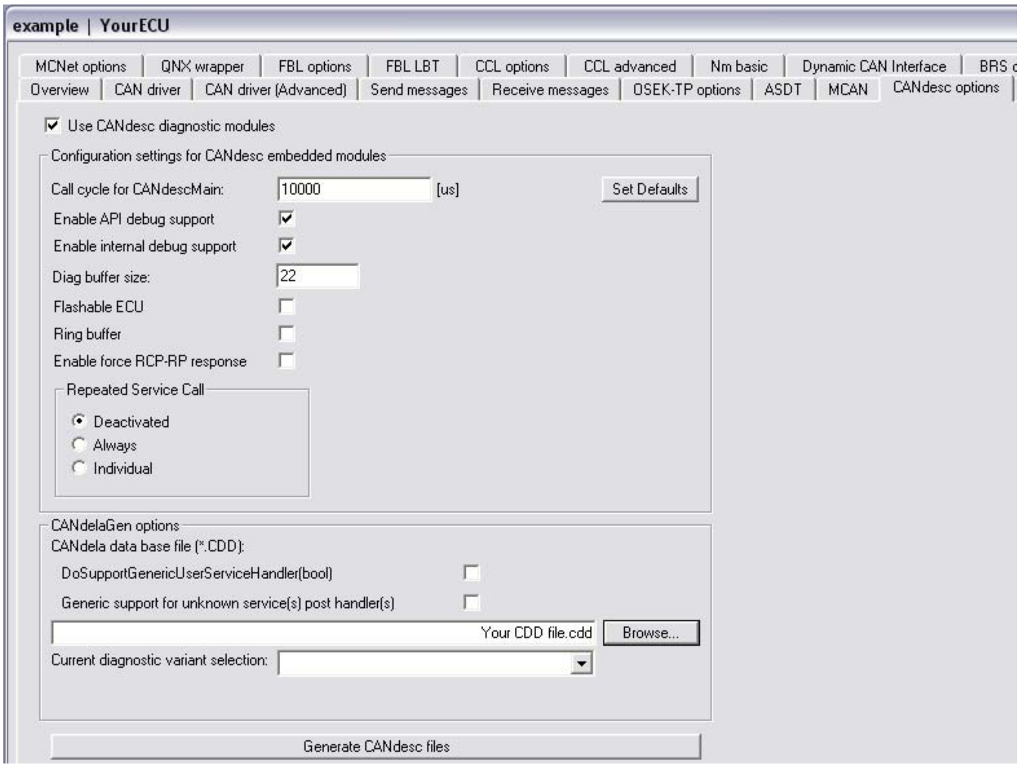
Make all the component settings as described in the appropriate User Manuals. For the Transport Protocol use the default **[Set Defaults]** for the first attempt.

correct path?

Info: Remember to set the paths where the generation tool does the output.

To configure CANdesc, open the **CANdesc options** tab. For this first attempt click **[Set Defaults]**. The generation tool needs to read an additional data base, the CANdela data base (CDD file). Browse for the CANdela data base file and select the CDD file you received from your vehicle manufacturer.

Very few settings have to be made in the Generation Tool CANgen for CANdesc



If the two checkboxes for debugging are checked you have to provide debug callback functions in your application.

A very important entry is the Call Cycle. This call cycle must be the one you call the DescTask function or the DescTimerTask function in your application (this will be explained in detail in the next steps).

4.5.2 Using the Generation Tool GENy

Open the generation tool GENy and create a new project as described in the OnlineHelp of GENy in the chapter **First Steps**.

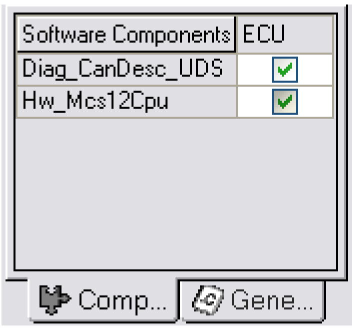
Info: Normally you get a data base (DBC) from your vehicle manufacturer that is designed for your project.

Make all the component settings as described in the appropriate User Manuals.

Info: Remember to set the paths where the generation tool does the output.

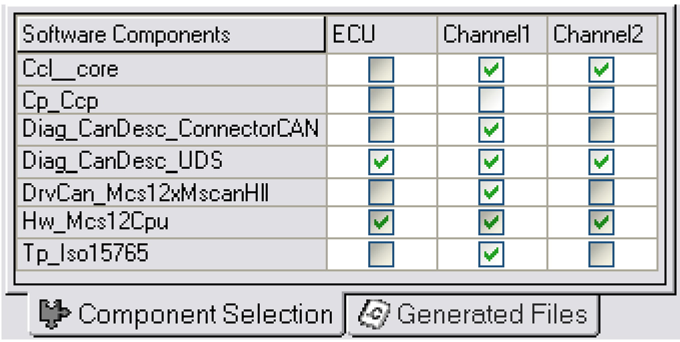
Activate the component CANdesc in the component selection view.

Component Selection View of GENy



The activation of the CANdesc component is modified with the Diag_CANdesc_xxx.DLL version 3.0.

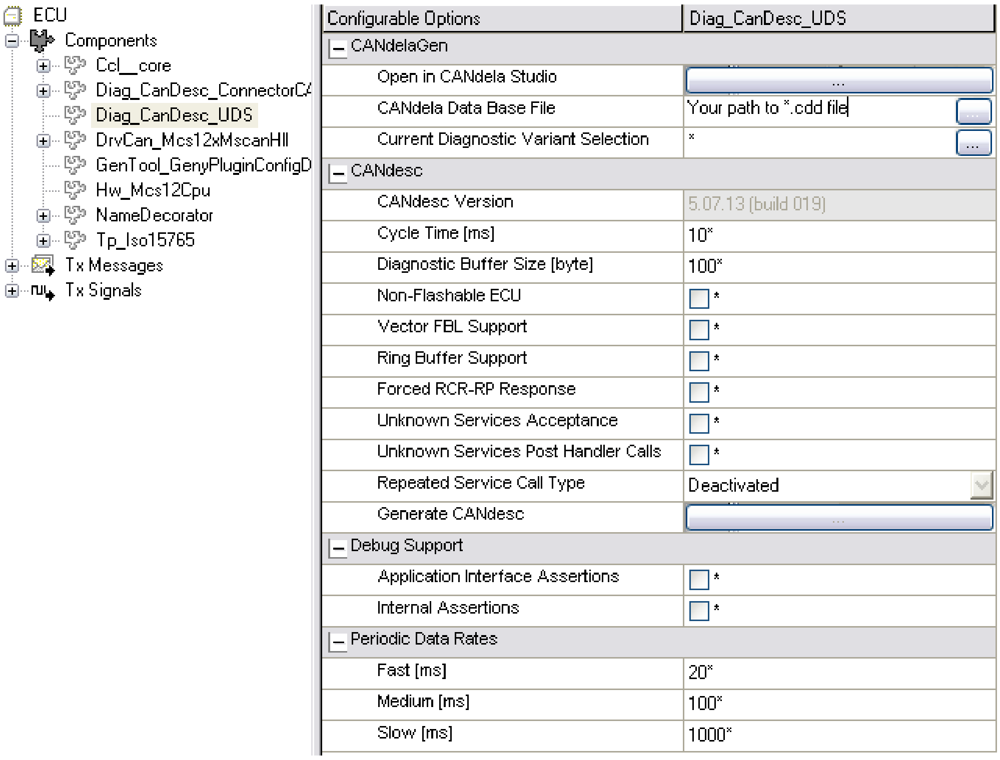
Component Selection View of GENy with separate CANdesc_Connector CAN component



Starting with this version CANdesc can be connected to more than one channel or can be used standalone. The Diag _CANdesc_UDS/KWP component includes the main configuration window of CANdesc. The Diag_CANdesc_ConnectorCAN component connects CANdesc to a CAN network and configures the TPMC to work with CANdesc.

Caution: If you do not activate CANdesc_ConnectorCAN component CANdesc will generate successful as standalone CANdesc. Therefore it is necessary to connect CANdesc with the CANdesc_ConnectorCAN component to a channel, if the TPMC shall be used.

GENy Configuration View for CANdesc



To configure CANdesc, open the **CANdesc** configuration via the Diag_CANdesc_UDS in the navigation view. As you see in the figure above, the generation tool needs to read an additional data base, the CANdela data base (CDD file). Browse for the CANdela data base file and select the CDD file you received from your vehicle manufacturer.

If the two checkboxes for **Debug Support** are checked you have to provide debug callback functions in your application.

A very important entry is the Call Cycle ("Cycle Time"). This call cycle must be the one you call the DescTask function or your DescTimerTask function in your application (this will be explained in detail in the next steps).

4.6 STEP Generating Files

4.6.1 Using Generation Tool CANgen

If you have finished the settings in the previous step, hit the **[Generate]**  button. You get a message box containing information about the generation process and a **[Success]** window containing information about the generated files and their paths. Check to see if the files are generated into the correct folders.

Success Window after a Generation Process

Open the folder you generated in the files listed above. There you should find the generated files for CANdesc, too. These are:

desc.c

This file contains the implementation and the private interface of the Diagnostic Software Component.

desc.h

This file contains the public interface of CANdesc. You will also find the <Negative response codes> here.

appdesc.c

This file is an implementation example for the proper usage of the diagnostics callback functions. All necessary callback functions are generated in this file and commented what is left to be done (<<TBD>>). See the example below:

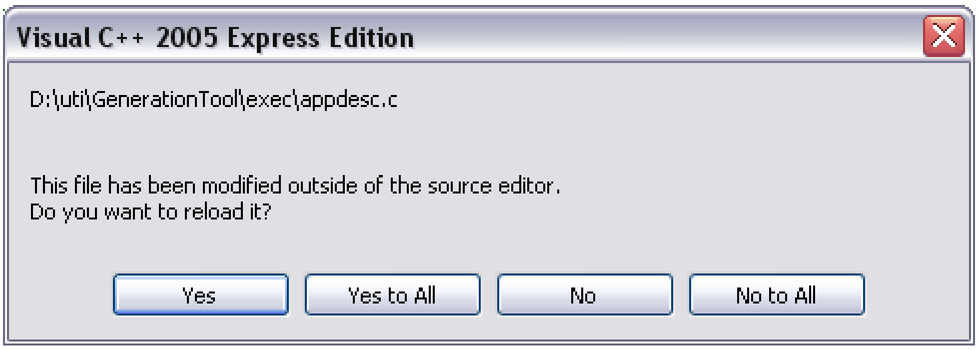
Example: Extract of the Generated Callback Functions Template.

```
/* ********************************************************************************
 * Function name:ApplDescReadVoltageService_Instance_For_Demonstration_Purposes
 * Description: Reads a signal.
 * Returns: signal value
 * Parameter(s): none
 * Particularitie(s) and limitation(s):
 *    - The function "DescProcessingDone" may not be called.
 *    - The function "DescSetNegResponse" may not be called.
 ******************************************************************************** */
vuint8 DESC_API_CALLBACK_TYPE
  ApplDescReadVoltageService_Instance_For_Demonstration_Purposes(void)
{
  /*<<TBD>> Remove this comment once you have completely implemented this function!!!*/
  /*Return the signal value.*/
  return 0xFF;
}

/* ********************************************************************************
 * Function name:ApplDescReadCurrentService_Instance_For_Demonstration_Purposes
 * Description: Reads a signal.
 * Returns: signal value
 * Parameter(s): none
 * Particularitie(s) and limitation(s):
 *    - The function "DescProcessingDone" may not be called.
 *    - The function "DescSetNegResponse" may not be called.
 ******************************************************************************** */
vuint8 DESC_API_CALLBACK_TYPE
  ApplDescReadCurrentService_Instance_For_Demonstration_Purposes(void)
{
  /*<<TBD>> Remove this comment once you have completely implemented this function!!!*/
  /*Return the signal value.*/
  return 0xFF;
}

/* ********************************************************************************
 * Function name:ApplDescReadResistanceService_Instance_For_Demonstration_Purposes
 * Description: Reads a signal.
 * Returns: signal value
 * Parameter(s): none
 * Particularitie(s) and limitation(s):
 *    - The function "DescProcessingDone" may not be called.
 *    - The function "DescSetNegResponse" may not be called.
 ******************************************************************************** */
vuint16 DESC_API_CALLBACK_TYPE
  ApplDescReadResistanceService_Instance_For_Demonstration_Purposes(void)
{
  /*<<TBD>> Remove this comment once you have completely implemented this function!!!*/
  /*Return the signal value.*/
  return 0xFFFF;
}
```

Appdesc modification

Detection to prevent loss of changes

If you start programming in the file appdesc.c, you fill in the missing code for the services and you start a new generation process, the generation tool detects whether the file has been changed or not:



Info: So better rename the file before you implement the diagnostic services.

appdesc.h

This file provides prototypes of the application diagnostic callback functions and

All callback function prototypes are generated in appdesc.h.

external application declarations, which are accessed by CANdesc.

Appdescdev.c

This file contains the definition of the used variables in CANdela Studio.

Info: This file shall be used only during the first integration in order to make your project fully compile- and linkable. This file is no necessary later, since the variables that will be defined here shall be implemented within your ECU application code.

Cross reference: (see section How to Handle Predefined Handlers (for MainHandler only) on page 38)

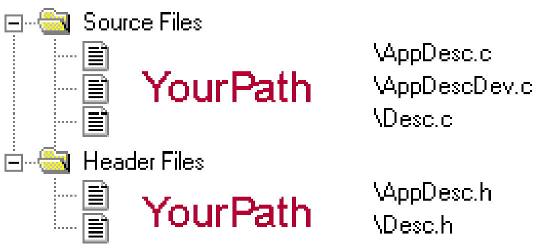
4.6.2 Using the Generation Tool GENy

If you have finished the settings in the previous step, hit the **[Generate]** button.

Info: All files for the CANdesc Software Component are generated!

Generated Files for CANdesc –CANdesc Core Files are Generated, too!

In the Generated Files view you see the files listed as shown in the figure below. Use this output to check the paths. In the list you only see the CANdesc-relevant files. The files are the same as generated with CANgen, so refer above for detailed information.



4.7 STEP Add CANbedded to your Project

What to do in this step depends on your development environment. Perhaps you are working with a makefile?

Regardless of this you have to add the CANbedded files to your project. These are the files discussed in Section Extract CANbedded Software Components on page 26 and the ones generated in the previous step.

Caution: Always make sure that the path in which you generate the files and the path your compiler is working on are the same!

Now there are several adaptations for you to make in your application.

4.8 STEP Adapt Your Application Files

Now all files for CANbedded and CANdesc are included in your project, and we can go on to make the necessary adaptations in your application files.

These adaptations can be split in two categories:

- Include, initialize and make the cyclic calls for the CANbedded Software Components (use the component-specific documentation for details).
- Connect your application to CANdesc

4.8.1 Including, Initializing and Cyclic Calling

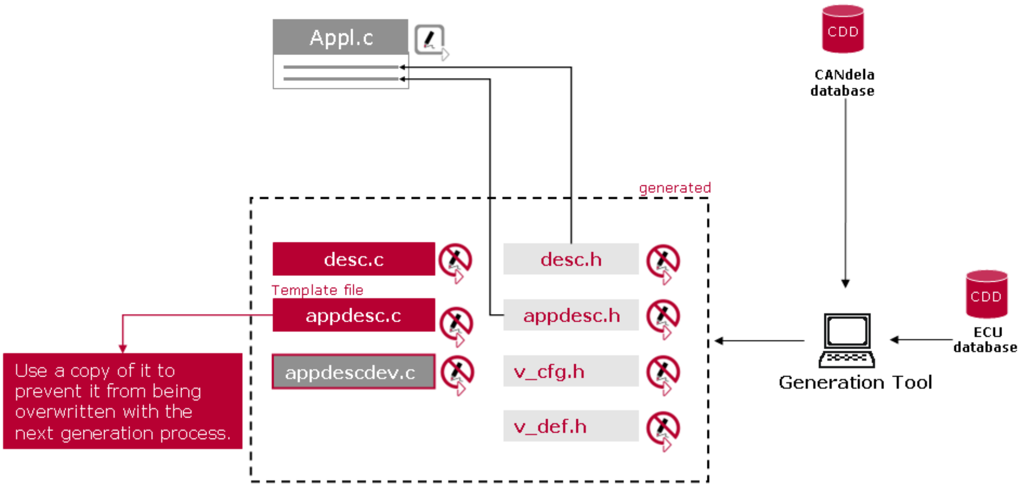
Two CANdesc headers have to be included in your application:

desc.h

appdesc.h

Keep the including file structure.

As for all other CANbedded Components, CANdesc must be included, initialized and used via a cyclic call.



The figure shows all generated files of CANdesc. Your application only needs to include the files desc.h and appdesc.h in the order they are mentioned.

Info: Any User Manual dealing with our CANbedded Software Components shows this kind of illustration. Always keep the include file structure that is shown.

Like all other CANbedded Software Components, CANdesc must be initialized and the Interrupts must be disabled during initialization

As for all other CANbedded Software Components the initialization function follows the same naming conventions. For CANdesc use:

```
DescInitPowerOn( initParameter );
                        /*Interrupts must be disabled*/
```

Cross reference: For information about the initParameter refer to your OEM-specific Technical Reference for CANdesc.

Make sure that DescInitPowerOn is called after the call of CanInitPowerOn and

TpInitPowerOn.

Normally the components are initialized from the bottom up according to the layer model. Always do these initializations with disabled interrupts.

This is the correct order of initialization if you use CAN Driver, Transport Protocol and CANdesc.

1. CanInitPowerOn();
2. TpInitPowerOn();
3. DescInitPowerOn(0);

As you adjusted things in Using the Generation Tool CANgen on page 26 (**Call cycle for CANdescMain**) the components need a cyclic call in your application to work properly. The call cycle must be the same as entered on the CANdesc tab / view (CANgen / GENy). The functions to call cyclically are:

It is very important that you call DescTask( ) or DescTimerTask() cyclically and keep the adjusted call cycle

```
DescTask( ); or
DescTimerTask(); (together with DescStateTask)
```

Caution: Never use DescTask() and DescTimerTask() / DescStateTask() together!

Using `DescTask`

With the call of this single function the handling of the timers and of the internal service processing (including application functions) is triggered. If you receive a diagnostic request, the response can be handled not until the next call of DescTask.

Info: This could lead to slower service processing.

Using `DescTimerTask` and `DescStateTask`

This concept splits the timer handling for CANdesc from the internal service processing. Now only the function DescTimerTask() has to be called in the predefined (configuration tool) cycle time.

The function DescStateTask() has to be called in a cyclic manner too, but does not need a fix cycle time. It can be called very fast to speed-up the reaction on a diagnostic request or it can be called as soon as there are free resources (e.g. an idle task in an operating system).

Info: CANdesc and DescTimerTask use the cyclic call as a time base for the timing calculations.

Do not make this call out of a timer interrupt. Just call DescTask() or DescTimerTask() at the task level.

4.9 STEP Functional Connection between your Application and CANdesc/CANdela Studio

It is up to you when you perform this step: before STEP Configuration with the Generation Tool (page 26), as a part of STEP Adapt Your Application Files (page 33) or perhaps at both times.

Info: There is a very close connection between the settings in CANdela Studio and what to do in your application.

Have a look a look at section Generic Handling of a Diagnostic Request in the CANdesc Component on page 21.

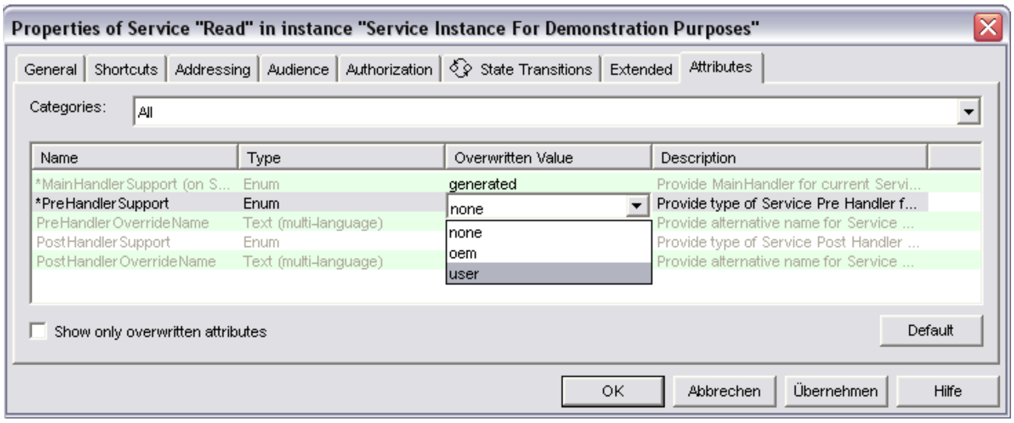
As you can see, there are three types of handlers (Pre-, Main- and PostHandler) that can be selected for any service. It is very important to know what happens when you choose the **Value** for the handlers. For this decision you need an overview of the great flexibility arising with the choice.

We will first go through the possible settings for one service as an example. With the knowledge you gain from this you can then go on with the other services.

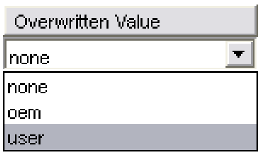
The settings of the handlers value can be made in the Properties windows of each service on the **Attributes** tab (see values in the following figure).

How are the settings in CANdela mapped to your application?

Support for the different Handlers can be adjusted on the Service Property Page



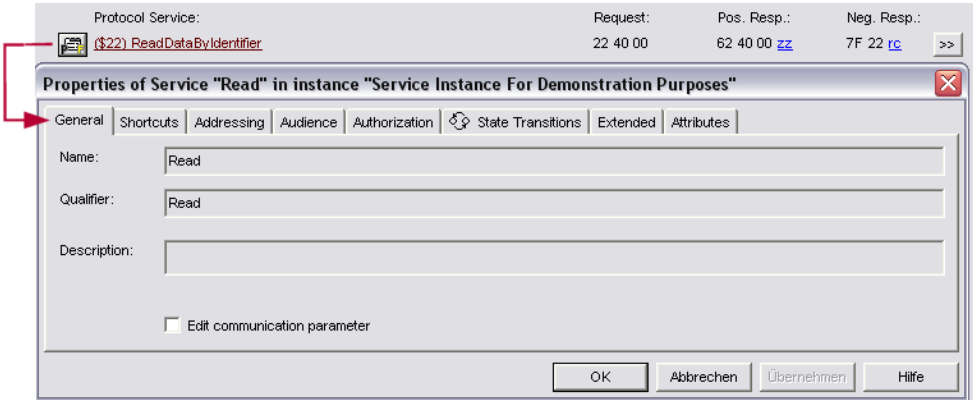
4.9.1 How to handle User-Defined Handlers



If you choose for the handlers to be user-defined, you have to do all the programming work for this service yourself, except for the checks. A callback function prototype will be generated in the file appdesc.h.

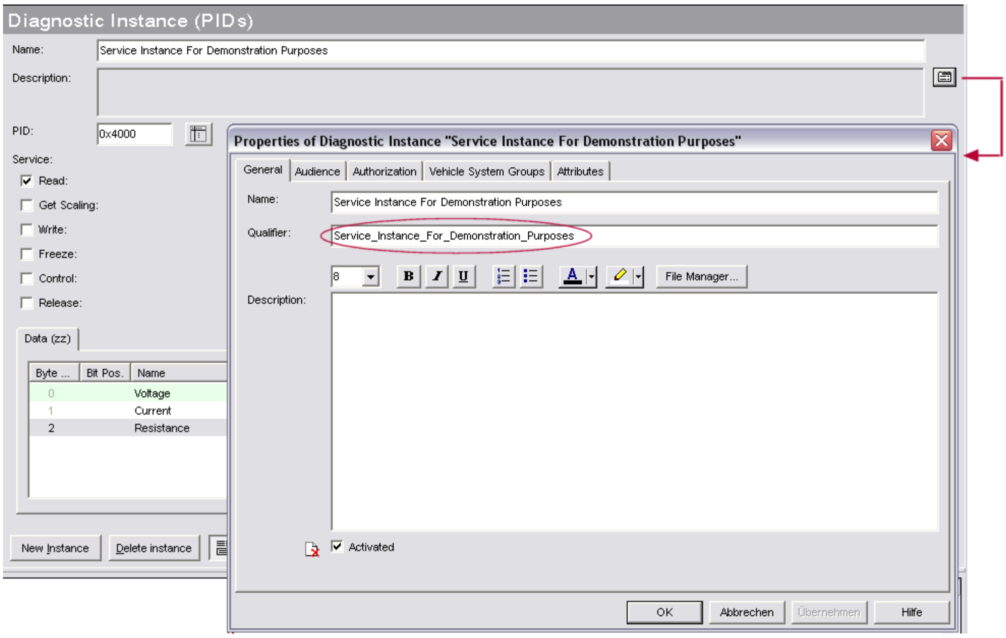
Service Qualifier

Open the Service Properties and then the **General** tab.



Diagnostic Instance Qualifier

Open the Diagnostic Instance Properties and then the **General** Tab



Names of the generated callback functions

The names of these callback functions are built as the following

Example: For this example, the callback function would look like this:

appldesc + **Read** + **Service_Instance_For_Demonstration_Purposes**

appldesc +**Pre**+ **Read** + **Service_Instance_For_Demonstration_Purposes**

appldesc +**Post**+ **Read** + **Service_Instance_For_Demonstration_Purposes**

with parameters:

```
void ApplDescReadService_Instance_For_Demonstration_Purposes(DescMsgContext* pMsgContext);
void ApplDescPreReadService_Instance_For_Demonstration_Purposes(void);
void ApplDescPostReadService_Instance_For_Demonstration_Purposes(vuint8 status);
```

Now you have to provide all the prototypes of the appdesc.h file as functions in your application and do the coding for each service, i.e. for each Pre-, Main- and PostHandler that is switched to User.

See an example for a ReadDataByIdentifier MainHandler for the service above defined for **User**. The data bytes of this service are:

- → g_Voltage (1 Byte)
- → g_Current (1 Byte)
- → g_Resistant (2 Bytes)

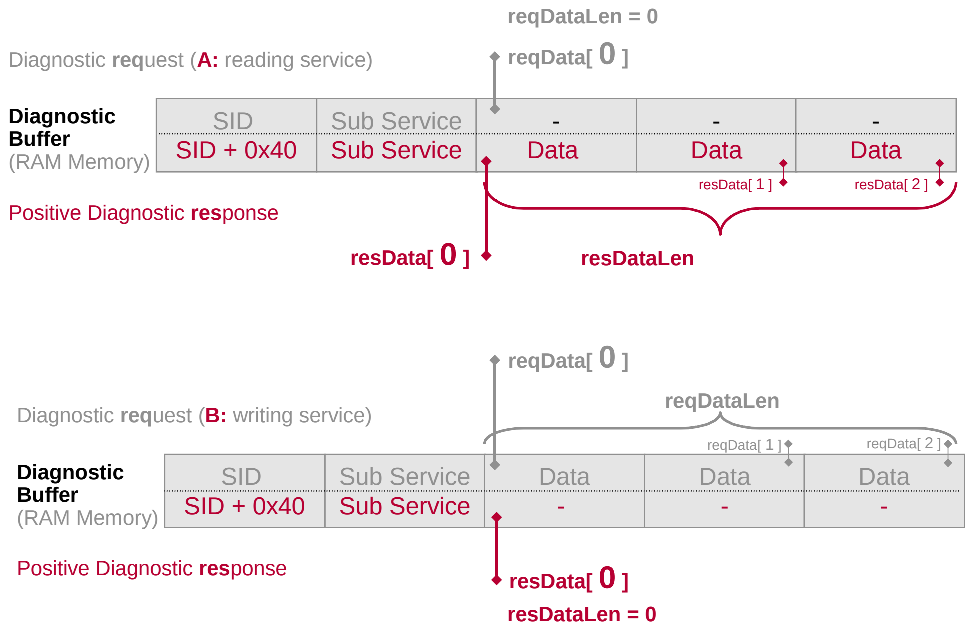
To process this service by yourself, you need to know how to access the diagnostic data. The following figure shows the data access for a reading service (upper figure) and a writing service.

A reading service consists of a SID and perhaps a Sub-Service. The requested data is then sent with the response.

A writing service consists of a SID, perhaps a Sub-Service and the data. The response is only a confirmation with SID+0x40 and perhaps a Sub-Service.

When working with CANdesc you only need to process the data. That is the reason why the pointer is directed to the first data byte.

The same Diagnostic Buffer is used for receiving a diagnostic request AND sending the response



Info: The request data and the response data are stored to the same memory location. Writing the response data means deleting the request data.

Example: The example below shows a very easy way to process a diagnostic request. The data is copied to the **Diagnostic Buffer**, the amount of the response data is determined and the diagnostic service is finished via DescProcessingDone.

Code Example for the MainHandler Using the User Option

```
void ApplDescReadService_Instance_For_Demonstration_Purposes(DescMsgContext* p
{
  pMsgContext -> resData[0] = g_Voltage;          /* First Signal g_Voltage */
  pMsgContext -> resData[1] = g_Current;          /* Second Signal g_Current */
  pMsgContext -> resData[2] = g_Resistance_lo;    /* The byte order depends on t
  pMsgContext -> resData[3] = g_Resistance_hi;    /* used byte format */
  pMsgContext -> resDataLen = 4;                  /* 4 data bytes */
  DescProcessingDone();
}
```

Example: When preparing the diagnostic response, it is very important to provide the correct data and calculate the length of the response (→resDataLen).

To finish the service processing with a positive response, call:

```
DescProcessingDone();
```

For a negative response, finish the service processing with:

```
DescSetNegResponse(<errorCode>);
DescProcessingDone();
```

Info: A negative response can also be set in the PreHandler. There it is enough to call DescSetNegResponse(<errorCode>). The PreHandler **must not** be finished with DescProcessingDone. See desc.h for the definitions of the error codes.

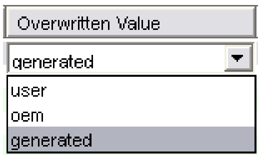
Remember: in the PreHandlers no access to the diagnostic data buffer is possible.

Response pending will be sent automatically by CANdesc

What to do if the response cannot be sent immediately?

In some cases (e.g. writing data to the EEPROM) you cannot send the response immediately, but you need not treat this as an exception. CANdesc will automatically inform the tester about the delay in the diagnostic response. So process the request and if you finish it, send DescProcessingDone. All other timing aspects are realized by CANdesc (Response Pending).

4.9.2 How to Handle Predefined Handlers (for MainHandler only)

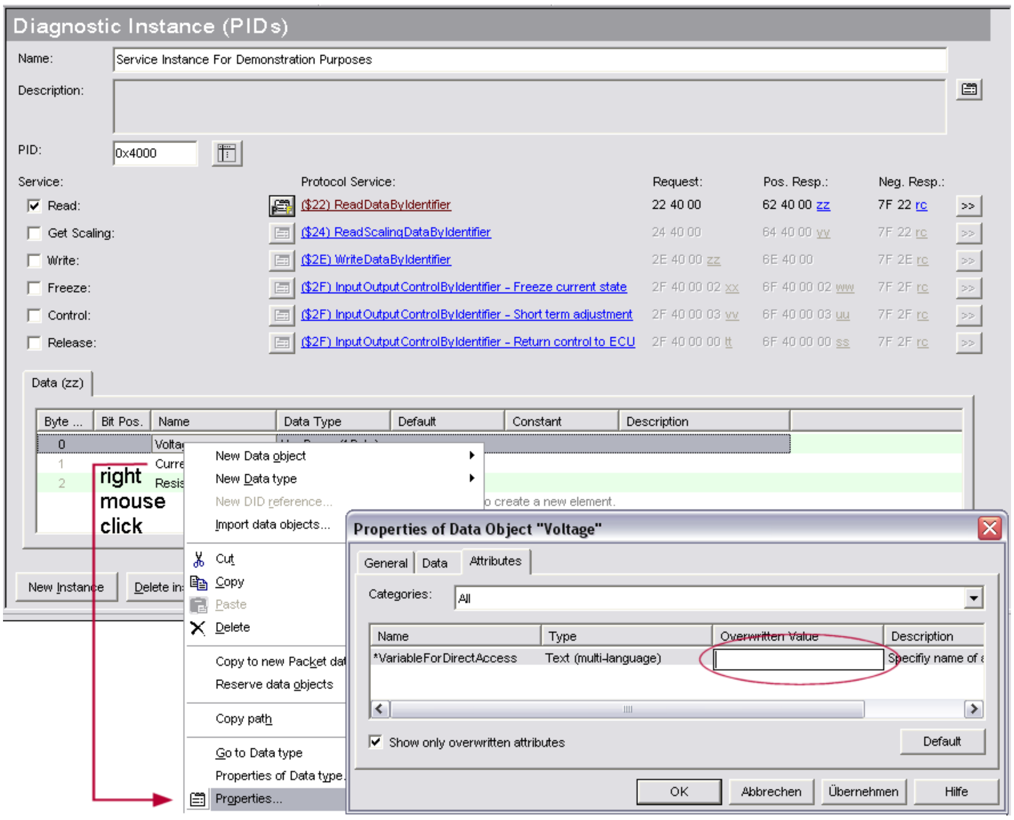


If you select **generated** you need not to program the complete service by hand. Using this option gives you two further options:

1. A signal callback function will be generated
2. You can tell CANdela the name of the variable (and data type) for a certain service and you only have to provide this variable in your application code.

To get a signal callback function generated, i.e. to implement the first option, right click on a data object and choose Properties from the pull down menu. Now the Properties window of the chosen data object opens. In this example it is the data object Voltage.

Signal Access via the Application and a Callback Function



Example: Make sure that the **Overwritten Value** field on the **Attributes** tab is empty. The generated prototype should look like this.

```
vuint8
ApplDescReadVoltageService_Instance_For_Demonstration_Purposes(
void);
```

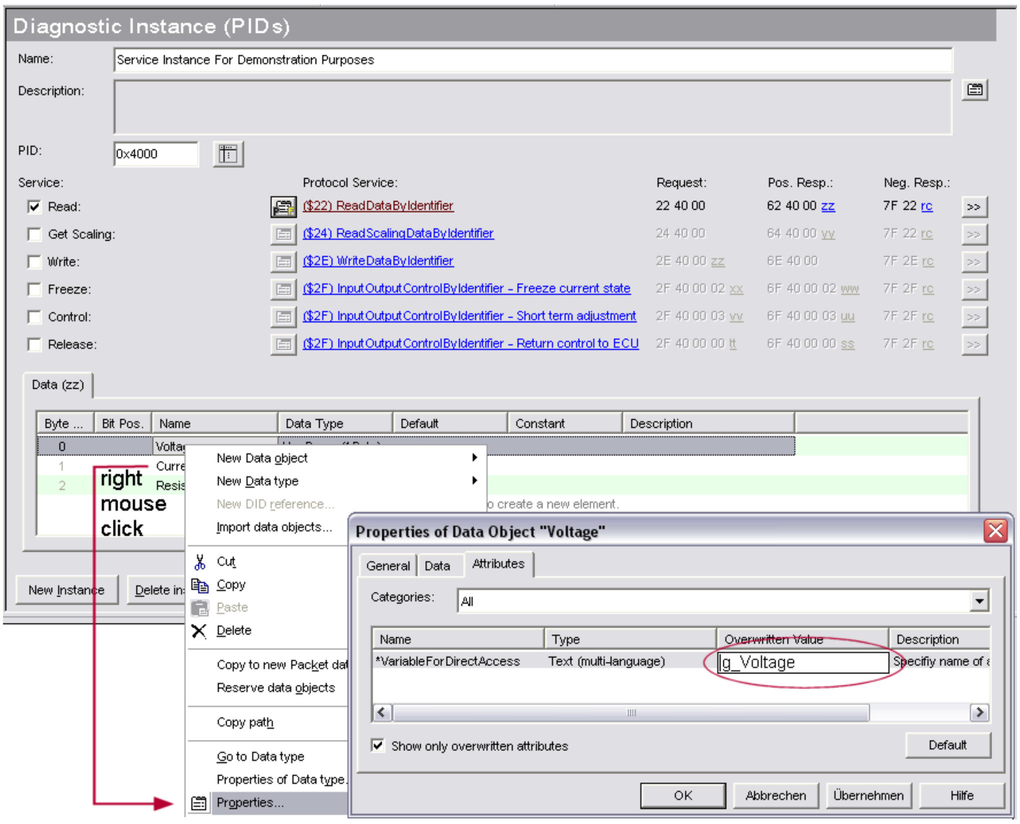
Example: All you have to do in your application for this MainHandler is to provide the function ApplDescReadVoltageService_Instance_For_Demonstration_Purposes and return the current value for the voltage stored anywhere in your application. The data type of the return value will be adjusted automatically to the data type (Element Type) in CANdela Studio. In this case it is a 1 byte value, therefore it is the data type vuint8.

```
vuint8
ApplDescReadVoltageService_Instance_For_Demonstration_Purposes(
void);
{
  return g_Voltage;
}
```

Generated does not mean that you do not have to do anything – but there is little programming work left to do

The second option is to connect the settings in CANdela Studio more closely to your application. Do the same steps as described above, but now enter the name of the variable in the value field of the Attributes tab as shown in the following figure.

Direct Signal Access



Example: Now an external declaration of the variable g_Voltage prototype should be generated.

```
extern vuint8 g_Voltage;
```

The data type for this declaration again depends on the element type of the data object, in this case 1 byte again.

Provide g_Voltage in your application (or use the appdescdev.c) and use it for storing the current voltage value. If a diagnostic request requests this value, CANdesc automatically refers to the content of g_Voltage. There is nothing more left to do for you.

4.9.3 Handling OEM-Specific Settings

The third choice is OEM. Do not change this. If the setting is on OEM, leave the settings as they are and refer to the OEM-specific documentation on how to deal with this service.

Now your task is to implement all diagnostic services you have to support and select the desired status for Pre-, Main- and PostHandlers (none, user, OEM, generated).

Caution: Do not touch the OEM-defined handlers.

Then save the settings. This will change the CDD file. Depending on which step you are on right now, either continue with STEP Configuration with the Generation Tool on page 26 or

start the generation process again to generated the files containing the changes you made.

Info: Sometimes in development, not all diagnostic services have been defined yet by the OEM. Provide this function anyway and send a negative response back. Then you can compile and link and test the other functions until the specification of the missing services is completed.

4.10 STEP Compile and link your Project

Now we have all the includes and all initializations. The components have the cyclic calls of their task functions and all callback functions are provided and programmed.

Start the compiler or makefile and get the project compiled and linked.

Is it ok? No errors?

Congratulations! That's it.

Go on to the next step and do the testing.

4.11 STEP Test it via CANoe

Since you have arrived at this step, you are now able to compile and link. Have you already downloaded the code to your target platform?

Testing of the generated CANdesc depends on you and the OEM you are working for. Perhaps you do have a diagnostic tester, perhaps not.

If you do not have an appropriate tester, we recommend using **CANoe** (a Vector PC tool) and one of its demo configurations.

4.11.1 Start CANoe.CAN OSEK TP enlarged

The CANoe demo environment is very simple way to basically test requests and responses

To test you diagnostics layer use one of the CANoe demo applications. Open this configuration via **Start/Programs/CANoe/Demos/More Demos/CANoe.CAN OSEK TP enlarged**.

A CANoe configuration will open with four nodes (A to D). All nodes look quite the same like this:

Set the baud rate in CANoe to the one of your ECU and connect it to CANoe via CAN (CANcardXL, CANAC2…). Now run CANoe via the yellow lightning bolt and run YourECU.

Info: Make sure that the CANoe mode is switched to **Real bus** and you have selected the same baud rate as the real node “YourECU” is working with.

4.11.2 Test of CANdesc

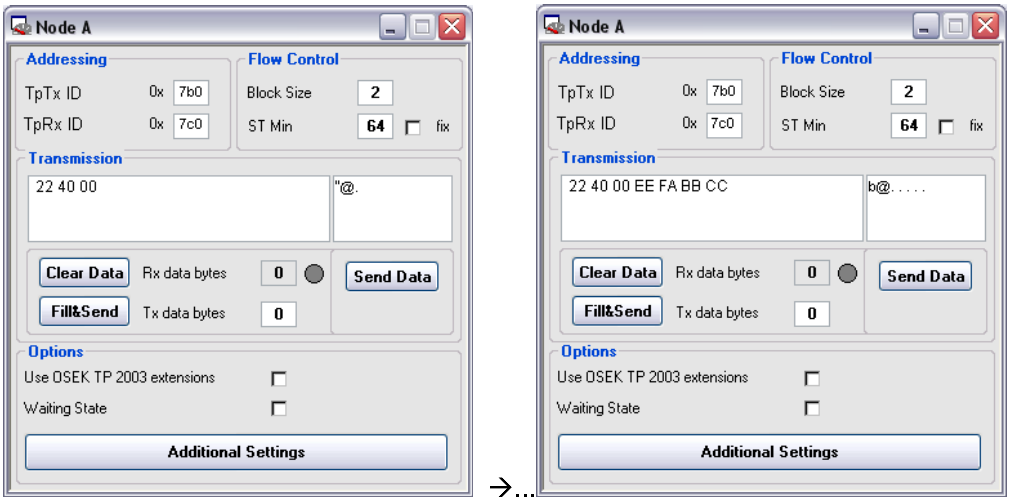
Use one of the four nodes for your tests. Change the TpTxId and the TpRxId in the “Addressing” field of the node window.

Caution: The TpTxId is the Rx Diagnostic message in your generation tool and the TpRxId is the Tx Diagnostic message. In the example case the DiagResponse message is 0x7C0 and the DiagRequest message 0x7B0.

It is optional to set the time for ST Min from 64ms (default) to 20ms. This is to prevent the ECU from running in time out.

Panel to Test Diagnostics Layer

Compare the Values with the ones shown in CANdela Studio



It is very simple to test the services using CANoe. Enter the request in the **Transmission** box and press **Send Data** and see the response in the same box. Compare this response with the desired one in CANdela Studio. The contents of the signals depend on the application.

Info: Make some variations to the signal contents to confirm the tests.

Repeat this for all other services.

5 Further Information

In this chapter you find the following information:

5.1 Diagnostic State Handling using CANdela Studio

Executing a diagnostic service generally causes a state change in the electronic control unit. Some services may only be executed if the electronic control unit is in a particular state. For example, services that change critical data may only be executed if the electronic control unit is first switched into a “security mode” (for example with the specification of a numeric key).

CANdela Studio offers the opportunity to define and edit global states and state transitions for the services of a diagnostic instance. In addition, states can be combined into state groups.

5.2 Typical Examples of State Groups and States in an Automotive Environment

The sessions (which should already be predefined) are a very “famous” example of a state group. Any diagnostic session has its set of services that are executable while the ECU is in this session. There are basically three sessions, defined from the ISO:

- Default session – as the name says, this is the standard session
- Programming session – while the ECU is in reprogramming mode (flashing)
- Extended Session – session for e.g. the development phase, providing an extended amount of services

Another very easy example for state groups is the security access. The ECU must be set to a specific state to be able to do critical data manipulation, such as the flashing action mentioned above. For example, the states for the state group security access would be:

- Locked
- Access granted

We use this example to very basically explain the state concept of CANdela Studio.

Cross reference: For more detailed information about this topic refer to the CANdesc Technical Reference.

5.3 Creating and editing State Groups, States and Transitions

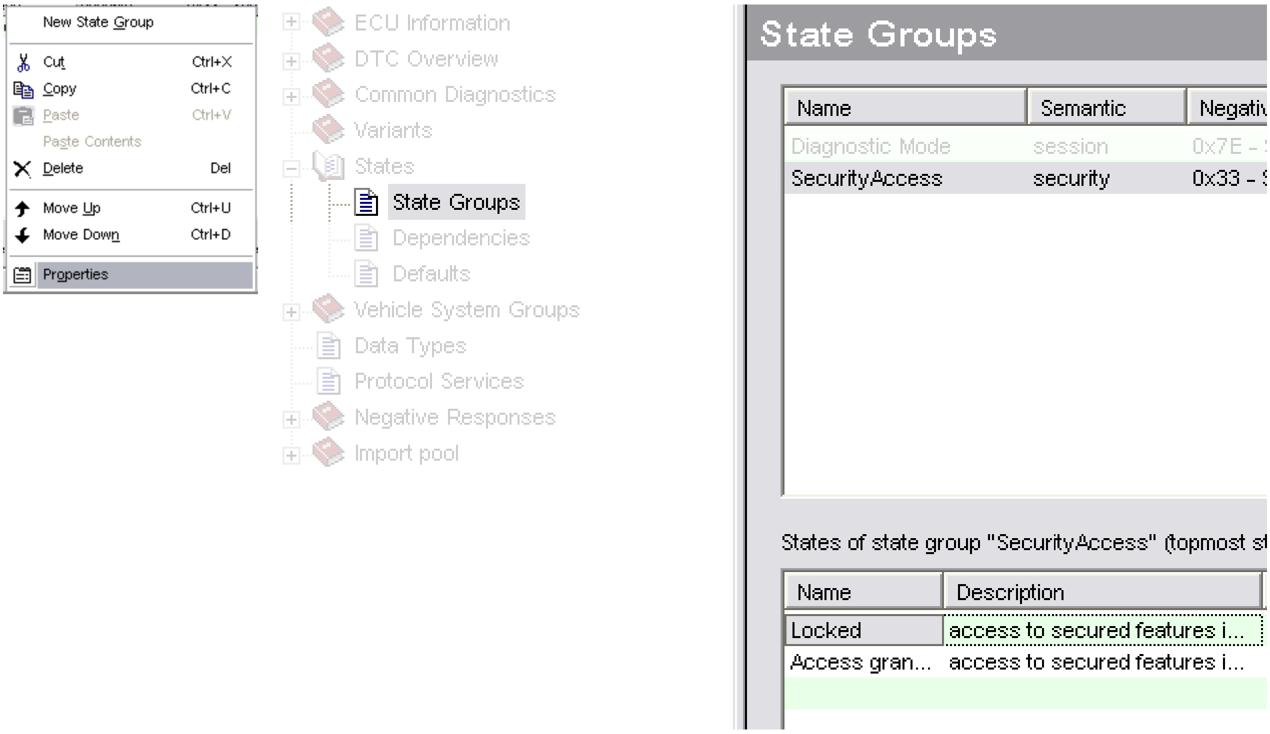
To create or edit the State Groups, click on [State Groups] in the CANdela Studio tree. Enter the new State Group Security Access by clicking on the text. A new State Group will be created called:

New State Group 1.

If you generate more than one State Group without renaming the previous ones, the groups are numbered counting from 1 up.

To edit the new State Group you have two options. The first is to click on the State Group name and edit the name, then click on the description field and enter the text. Another way is to open the pull down menu of the State Group with a right click on the row of **SecurityAccess** and select **Properties**. The **Properties of State Group Security Access”** window will open. Enter the name and description.

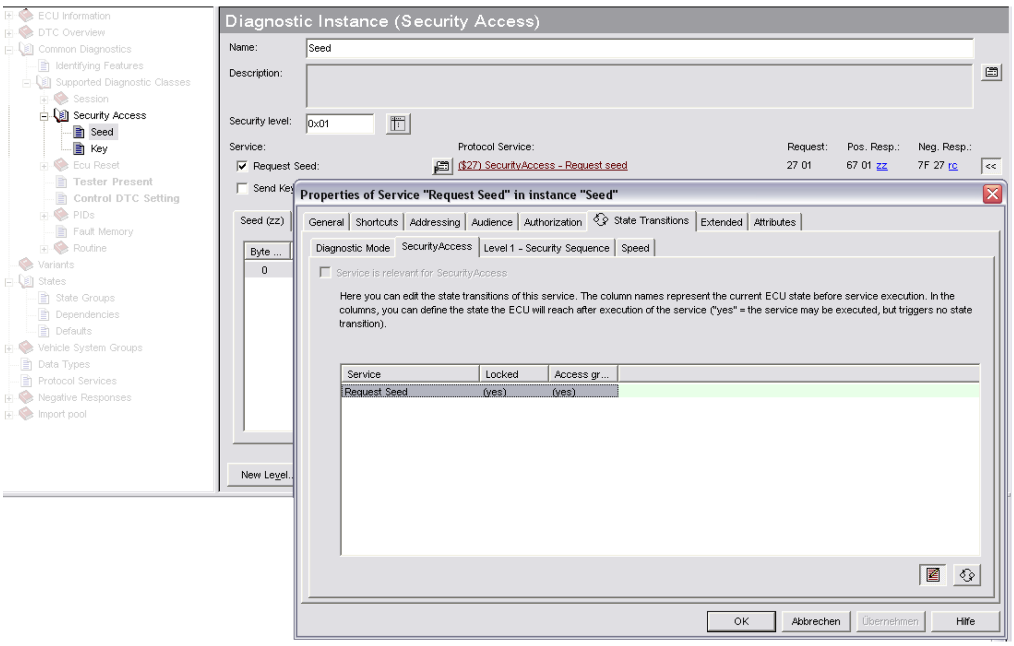
Info: The qualifier will be created automatically.



Now we can add the states below in the same way. Click on the text to create a new element, adjust the names and enter a description.

The next step is to assign the relevant services to the states.

Defining States for the Service SecurityAccess – Request Seed



Select the Diagnostic Instance Security Access Seed and open the **Properties** of the **Service Request Seed**. Select the tab **State Transitions** and then **SecurityAccess**.

You see the service with the two columns states **Locked** and **Access granted**.

Info: To select yes or no just select the row, click on the yes/no and then use the pull down menu.

Info: Pull down menu selections:

No = Must not be executed

Yes = may be executed, no state transition

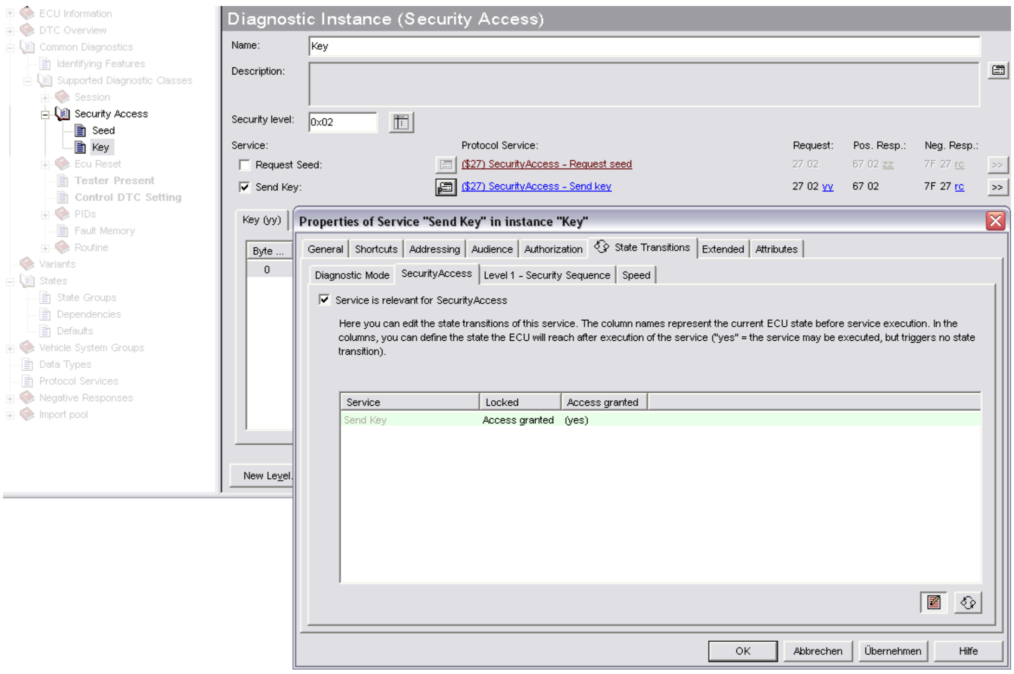
Locked = state transition

Access granted = state transition

The following figure shows the properties for the service **Send Key** in the Key instance. This service is also assigned to both of the states, but there is also a **transition to state** defined. How do you interpret this entry?

The service **Send Key** could be executed in the state **Locked**. If the data is processed (depending on the OEM, this must be done by the application or is a generated, OEM-specific Code) and a positive response is sent back, CANdesc switches the state from **Locked** to **Access Granted**. In case of a negative response the ECU remains in the diagnostic state Locked.

A positive response is the trigger for a transition from the Locked state to the state Access granted



5.4 Connection between the states and your application

The initial state after the ECU starts is the state at the top of the list. In this case the initial state is Locked.

Info: Think about the states very carefully before editing. Make sure that the initial state is listed on top.

Example: The state transition mentioned above is monitored to your application via a callback function. You will find the prototype of this function, as usual, in the appdesc.h file. It may look like this:

```
void ApplDescOnTransitionSecurityAccess(DescStateGroup
newState, DescStateGroup formerState);
```

The parameters show the direction of the transition. Provide the function and react to a transition as you wish.

Example: There is another way to switch states. Leave the transition to state empty and do the state transition in your application. This could look like:

```
DescSetStateSecurityAccess(
kDescStateSecurityAccessAccess_granted );
```

Use

```
DescStateGroup DescGetStateSession (void)
```

to find out the current session.

Info: The function declaration and parameter can be found in the generated file desc.h.

5.5 Diagnostic Buffer

As described in chapter How to handle User-Defined Handlers on page 35, the diagnostic buffer is an area in the RAM where the application and the CANdesc Software Component are allowed to write on and read from. How this is handled is described in this chapter above.

What is not explained until now is:

- → how to choose the length of the diagnostic buffer
- → that there are two mechanisms of using the buffer and
- → when to use which mechanism

5.5.1 Linear Diagnostic Buffer

The easiest way of using the diagnostic buffer is to use it as a linear buffer. The size of the buffer in bytes must be the size of the longest data (diagnostic response or request).

Info: Normally this is a diagnostic trouble code message (DTC) and can reach up to 100 bytes and more.

Copy the complete response information to the diagnostic buffer and confirm this via the call of `DescProcessingDone`.

This is easy to handle but there are some disadvantages arising with this concept:

- → The RAM consumption could be enormous
- → The delay time between the reception of a Diagnostic Request and the first response message could be very long, depending on the service and the amount of bytes of the response message.

There is another concept without these disadvantages but this concept needs a little bit more insight in CANdesc functionality.

5.5.2 Ring Buffer Mechanism

There are several reasons for using the ring buffer mechanism:

- → Little RAM consumption because of small diagnostic buffer
- → Shorter delay between the diagnostic request and the first response message

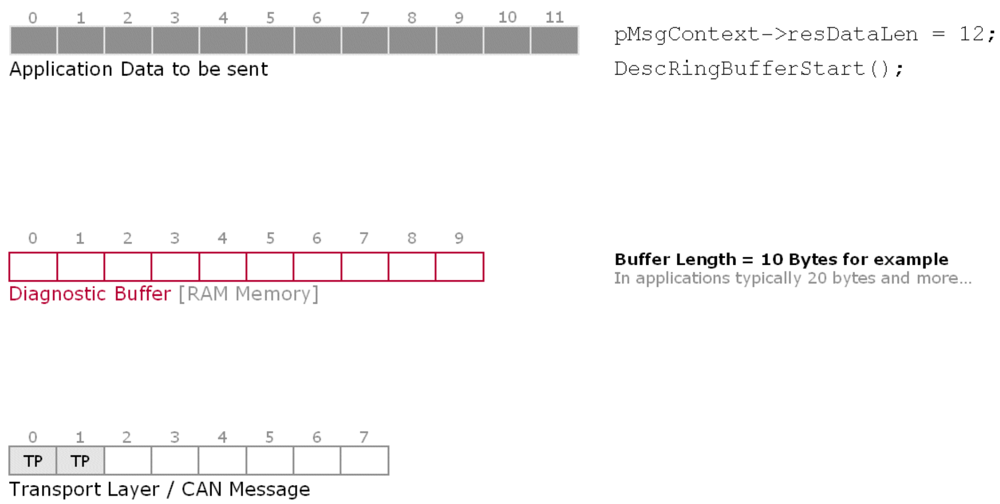
The ring buffer mechanism offers the following features:

- → Asynchronous writing of serial diagnostic data to the diagnostic buffer
- → Underrun allowed, time monitored (in case of TP underrun the PostHandler is called with a Tx error code)
- → Overrun prevented and monitored via return code

One of the advantages of the ring buffer mechanism is the little RAM consumption (compared with the linear buffer). The consequence is that this little diagnostic buffer can hold less data than a diagnostic buffer designed for linear buffer mechanism. That means that the application has to fill the buffer in portions until the complete diagnostic response is sent.

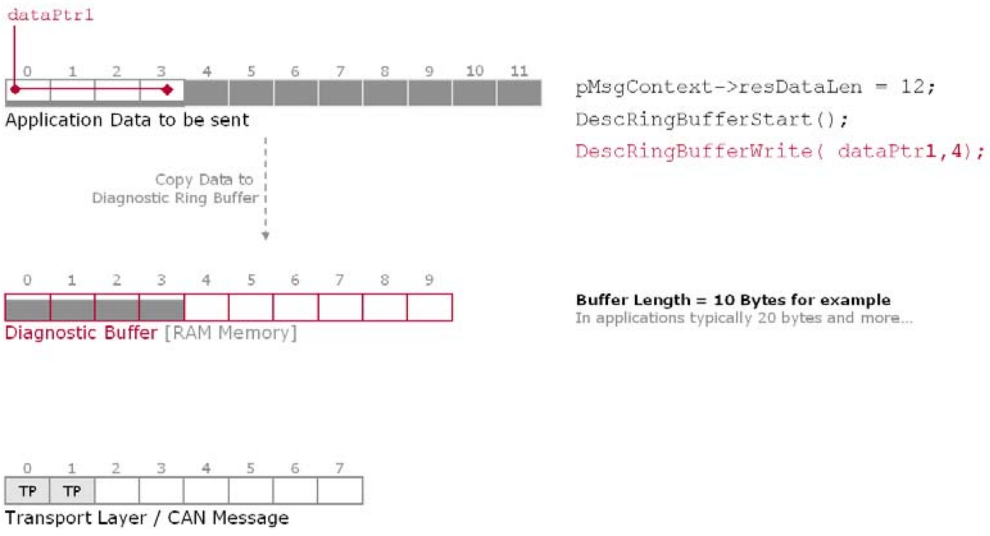
The following example is very simple and designed to understand the concept behind the ring buffer mechanism.

Ring Buffer STEP 1 – Application Data and Ring Buffer



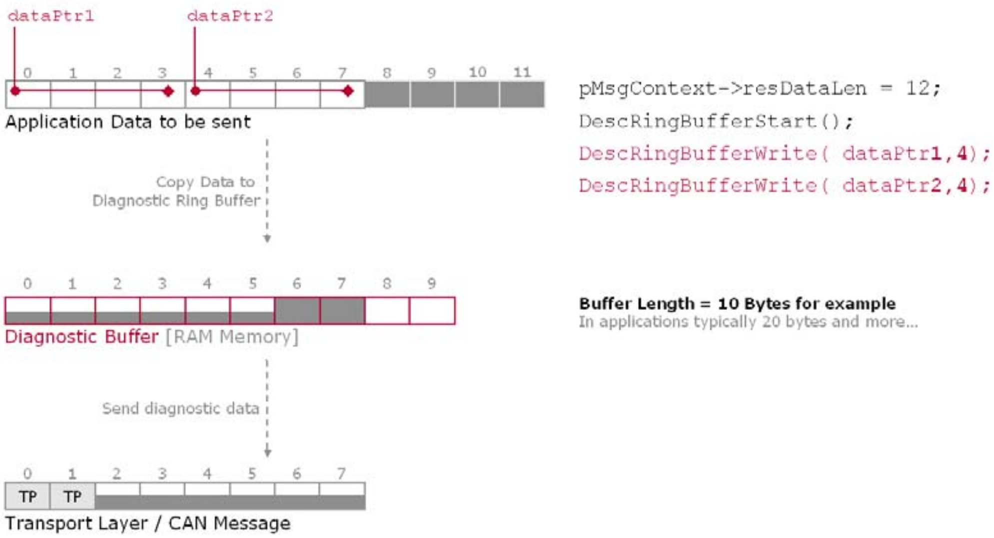
Starting point is a diagnostic buffer with 10 bytes size and 12 bytes of application data to be sent. First you have to set the length of the complete diagnostic data (`resDataLen = 12`) and start the ring buffer mechanism (`DescRingBufferStart`).

Ring Buffer STEP 2 – First four data bytes are copied to the Ring Buffer



Now hand over the pointer to the location of the first four application data bytes (pointer and amount of data - `DescRingBufferWrite`) to the CANdesc Software Component. CANdesc Basic copies the four data bytes to the diagnostic buffer.

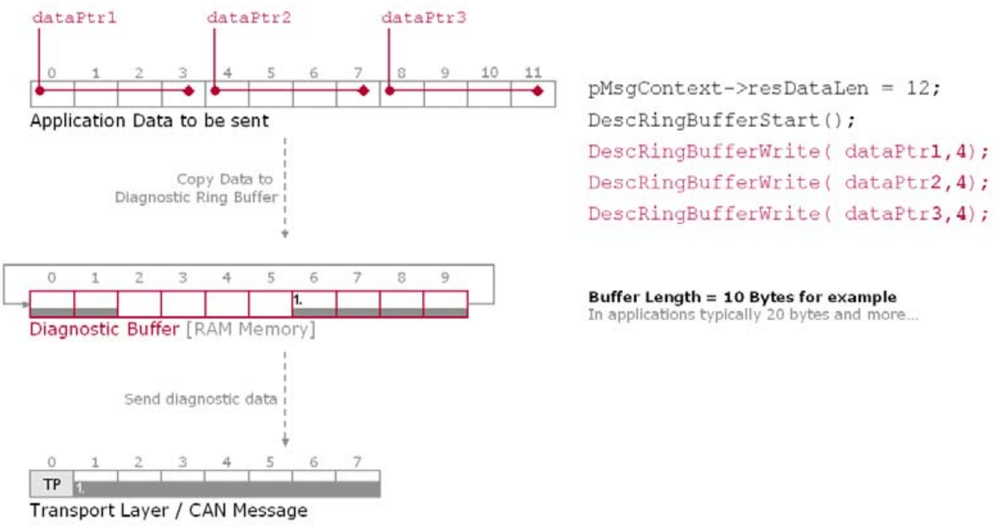
Ring Buffer STEP 3 – Eight Data Bytes in the Diagnostic Buffer, six Bytes are being sent via CAN



Hand over the pointer to the location of the next four application data bytes and CANdesc copies the data to the diagnostic buffer right **after** the first four bytes. Now there is enough data in the buffer and CANdesc sends the first six data bytes via the CAN bus.

Info: The first 2 bytes of the message are transport information and therefore not free for application data (TP bytes on position 0 and 1).

Ring Buffer STEP 4 – The Diagnostic Buffer is filled round robin



Now there are only four bytes left to be copied to the Diagnostic Buffer. The first two bytes are stored in position 8 and 9 of the buffer, the next two bytes in position 0 and 1.

Info: Now it should be obvious why this concept is called Ring Buffer; the buffer is filled round robin.

In a next step the six data bytes will be copied and sent via CAN starting with the byte on position 6.

That is the basic mechanism, but how do you know when there is enough space in the buffer? What happens if the application writes data and the buffer is not free? How to handle this buffer in code details?

5.5.2.1 Activation of the Ring Buffer

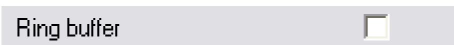
Activation of Ring Buffer in GENy

Although the ring buffer could be used for any service and you can meet this decision at run-time you must activate this functionality in general.

Do this on the CANdesc configuration view in GENy by clicking the **Ring Buffer Support** checkbox.

Activation of Ring Buffer in CANgen

In CANgen you have to select the **Ring buffer** checkbox at tab **CANdesc Options**.



5.5.2.2 Main Control Functions for the Ring Buffer Mechanism

Cross reference: For a more detailed description of the API refer to the TechnicalReference_CANdesc.pdf.

DescRingBufferStart

The call of this function starts the ring buffer mechanism. You can use it for any service and it replaces the `DescProcessingDone` that you use for the linear buffer mechanism.

Info: Call `DescRingBufferStart` on MainHandler level.

DescRingBufferWrite

Via this function you tell CANdesc the location and the amount of the application diagnostic data and the software component copies this data to the diagnostic buffer.

The function has two parameters; one is a pointer which points to the memory location of the next diagnostic data. The other parameter is the amount of data that should be copied (should be lower or equal to the ring buffer size).

The return value of this function can be `kDescOk` or `kDescFailed` and indicates that the write process to the diagnostic buffer was successful or that there was not enough free space in the buffer.

Info: In case of `kDescFailed` no data has been written to the diagnostic buffer.

DescRingBufferGetFreeSpace

This function shows the amount of free space in the diagnostic buffer.

DescRingBufferGetProgress

This function shows the amount of data that has already been written to the diagnostic buffer (for this service).

5.5.2.3 Examples for Ring Buffer Mechanism

Now start the coding for the example above (chapter 5.5.2). The diagnostic buffer is 10 bytes and the amount of application data to be sent via a diagnostic response is 12. In the example you write to the diagnostic buffer in four byte portions.

The examples use an OSEK-OS operating system, but it should be very easy for you to transfer this to a system without OSEK-OS.

Ring Buffer Example 1 - “Write and Check”

Example: MainHandler of the Service “Service”

```
uint8 state; /*global variable*/

void ApplDescService( DescMsgContext* pMsgContext)
{
  pMsgContext->resDataLen = 12;  /*amout of the complete data to be sent*/
  DescRingBufferStart( );
  state = 0;
  DescRingBufferWrite( &dataPtr[state*4], 4 );  /*first write to diagnostic buffer*/
  state++;
  SetRelAlarm(ALServiceStateMachine, 0, <cycle> ); /*Alarm for activating the Basic
                                                        TASK*/
}
```

Define the length of the complete diagnostic response (`resDataLen = 12`) and start the ring buffer mechanism (`DescRingBufferStart`). The global variable state is to identify in which state your state machine is and it is an index for the data pointer `dataPtr`.

In the MainHandler you write to the diagnostic buffer the first time for this service - it must be free. So you can write the first four data bytes via `DescRingBufferWrite`.

Info: As the handling of the diagnostic (CANdesc only works if its task is called cyclically) needs a cyclic call of the DescTask() or DescStateTask() you have to fill the diagnostic buffer gradually e.g. by the means of a cyclic basic task. Otherwise the DescTask() or DescStateTask() would not be called and the CANdesc could not work.

Now start an alarm to get the basic task `BTServiceStateMachine` called all <cycle> ms.

Basic Task to Handle the Service State Machine

```
TASK( BTServiceStateMachine )
{
  if( DescRingBufferWrite( &dataPtr[state*4], 4 ) == kDescOk )
  {
    state++;
  }
  if( state == 3 )
  {
    CancelAlarm( ALServiceStateMachine );   /*all data (3x4 bytes) has been
                                                 transferred to diagnostic buffer*/
  }
  TerminateTask( BTServiceStateMachine);
}
```

This basic task is designed to write the next 8 data bytes to the diagnostic buffer. But the application does not know if the buffer is free or not (**Write and Check**). To get this information use the return value of the `DescRingBufferWrite` function. Is it `kDescOk`, then the write was successful and we can increment the state. If not (`kDescFailed`), we have to repeat writing the last four bytes again in the next call of the task.

If `state` is equal to three, i.e. all 12 bytes have been written to the diagnostic buffer, we cancel the alarm to stop the handling of this diagnostic service.

Ring Buffer Example 2 - “Check and Write”

The MainHandler for this example is the same as in example 1.

The difference is, that you first check whether there is enough free space in the buffer before you write the next data (**check and write**). Via the function `DescRingBufferGetFreeSpace` you get the information about the free space in the buffer. If there is enough space, write the next data and increment the state, if not, terminate the task and repeat the try with the next activation of the task.

Example:

```
TASK( BTServiceStateMachine )
{
  DescMsgLen freeSpace;
  freeSpace = DescRingBufferGetFreeSpace();  /*MISRA*/
  if( freeSpace >= 4 )
  {
    DescRingBufferWrite( &dataPtr[state*4], 4 );
    state++;
  }
  if( state == 3 )
  {
    CancelAlarm( ALServiceStateMachine );   /*all data (3x4 bytes) has been
                                                transferred to diagnostic buffer*/
  }
  TerminateTask( BTServiceStateMachine);
}
```

Ring Buffer Example 3 – “GetProgress”

In this example you use the already mentioned function `DescRingBufferGetProgress` to figure out how many bytes you have written to the buffer until now. This makes the example much easier but a little bit more difficult to understand why it works in this way.

As you see you do not need a global variable for the state. The state now is defined by the amount of data that you have already written to the buffer.

Example:

```
void ApplDescService(DescMsgContext* pMsgContext)
{
  pMsgContext->resDataLen = 12;
  DescRingBufferStart();
  DescRingBufferWrite( &dataPtr[ DescRingBufferGetProgress() ], 4); /* will be 0 at
                                                                       the beginning*/
  SetRelAlarm( ALServiceStateMachine, 0, cycle ); /*Alarm for activating the Basic
                                                         TASK*/
}

TASK( BTServiceStateMachine )
{
  DescMsgLen progress = DescRingBufferGetProgress();
  if(progress < 12)
  {
    DescRingBufferWrite( &dataPtr[ progress ], 4 );
  }

  TerminateTask( BTServiceStateMachine );
}
```

Conclusion

As you see in these three little examples, the handling of the ring buffer is always the same. You start the writing, you write cyclically and in portions and you have to define an ending criteria – a typical state machine.

CANdesc offers a feature to support that kind of handling that is not only useful when working with ring buffer mechanism – the repeated service call.

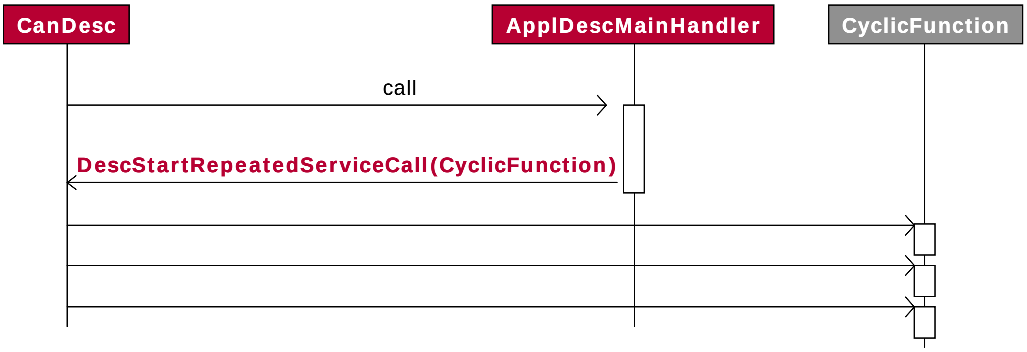
5.6 Repeated Service Call Feature

The easy way would be to transfer all data in the MainHandler to the diagnostic buffer, to call `DescProcessingDone` and the service is done.

But what to do with information that cannot be provided immediately? For this reason you have to trigger a further function that handles the provision of diagnostic data and then finishes the service via `DescProcessingDone`.

The Repeated Service Call helps you to handle situations like above very easy. Via the function call `DescStartRepeatedServiceCall( CyclicFunction )` you trigger the call of the "`CyclicFunction`" with the call cycle of `DescTask` or with the call of `DescStateTask`.

Repeated Service Call



The **CyclicFunction** can be the function where from you call the repeated service call or a second function.

At the end of the service handling you can stop the function from being called cyclically in two ways:

- call DescProcessingDone in linear mode
- if you have copied all announced data bytes to the diagnostic buffer if ring buffer mechanism is used

The repeated service call is stopped too, if you

- call DescRingBufferStart
- call (another) DescStartRepeatedServiceCall( )

Info: Using repeated service call and the ring buffer you have to take care about the order `DescRingBufferStart` and `DescStartRepeatedServiceCall`.

5.6.1 Activation of the Repeated Service Call

As the ring buffer mechanism you have to activate the repeated service call in the generation tool.

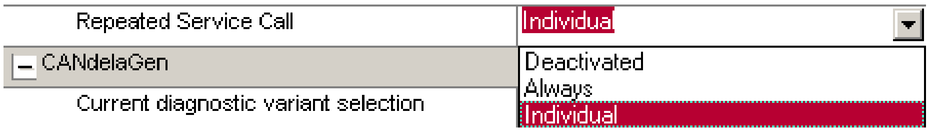
In GENy you have to select a mode for repeated service call in the CANdesc configuration view. CANgen offers the same modes in the CANdesc option tab.

As you see in the screenshot there are three modes for the Repeated Service Call:

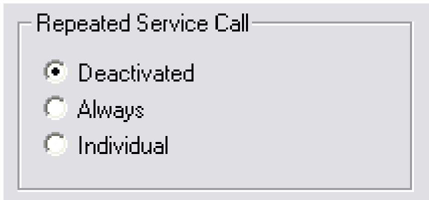
Deactivated	You cannot use this feature at all.

Deactivated	You cannot use this feature at all.
Always	The repeated service call is switched to on for any service in the way that the MainHandler is called cyclically as long as you call `DescProcessingDone` or all data is written to the ring buffer.
Individual	With the individual setting you decide for every service whether to use the repeated service call or not. To use it, just activate it via `DescStartRepeatedServiceCall` as you see in the following examples.

Selection for Repeated Service Call in GENy



Selection for Repeated Service Call in CANgen



The following two examples show the handling of the ring buffer mechanism using the repeated service call.

Info: The setting in the generation tool is individual.

5.6.2 Repeated Service Call and Ring Buffer 1 – "Write and Check"

This is the same example as in the chapter dealing with the ring buffer mechanism. This time use the repeated service call instead of the OSEK-OS task. And in this first example, define the MainHandler itself to be called cyclically via:

Example: `DescStartRepeatedServiceCall( ApplDescService );`

For this case the MainHandler must be realized as a state machine because the start of the repeated service call has to be done only once per diagnostic request handling.

Example:

```
uint8 state; /*global variable, set to 0 in PreHandler*/

void ApplDescService( DescMsgContext* pMsgContext)
{
  if( state == 0)
  {
    pMsgContext->resDataLen = 12;  /*amout of the complete data to be sent*/
    DescRingBufferStart( );
    DescRingBufferWrite( &dataPtr[ state*4 ], 4 )
    DescStartRepeatedServiceCall(ApplDescService);
    state++;
  }
  else
  {
    if( DescRingBufferWrite( *dataPtr[ state*4 ], 4 ) == kDescOk )
    {
      state++; /*if resDataLen data bytes have been copied to the diagnostic
                 buffer the repeated service call stops automatically*/
    }
  }
}
```

5.6.3 Repeated Service Call and Ring Buffer 2 – "Check and Write"

Now add a second function and call it cyclically after the MainHandler has been called. The MainHandler acts as initialization of the state machine and the second function handles all further states.

Example:

```
uint8 state; /*global variable*/

void ApplDescService( DescMsgContext* pMsgContext)
{
    state = 0;
    pMsgContext->resDataLen = 12;        /*amout of the complete data to be
sent*/
    DescRingBufferStart( );
    DescRingBufferWrite( &dataPtr[ state*4 ], 4)
    DescStartRepeatedServiceCall( SecondFunction );
}

void SecondFunction( DescMsgContext* pMsgContext ) /*prototype must be defined
                                                      by application*/
{
  DescMsgLen freeSpace;
  freeSpace = DescRingBufferGetFreeSpace();  /*MISRA*/
  if( freeSpace >= 4 )
  {
    state++;
    DescRingBufferWrite( &dataPtr[ state*4 ], 4 );
    /*if resDataLen (12) data bytes have been copied to the diagnostic buffer
      the repeated service call stops automatically*/
  }
}
```

6 Additional Information

In this chapter you find the following information:

6.1 Persistors

What is the Persistor for?

The CANdela data base file (CDD) is created by CANdela Studio and used by GENy for configuring CANdesc.

If you use a newer version of the CANdela Studio, the format of the CDD file could be also newer than your GENy is able to deal with.

The Persistors are responsible to convert the newer CDD file into a CDD file which is able to read by GENy.

Update Persistors – Download current Version

The latest Persistors can be downloaded from Vector homepage

www.vector.com.

Select **Downloads** and then the three settings for **Products**, **Categories** and **Standards**.

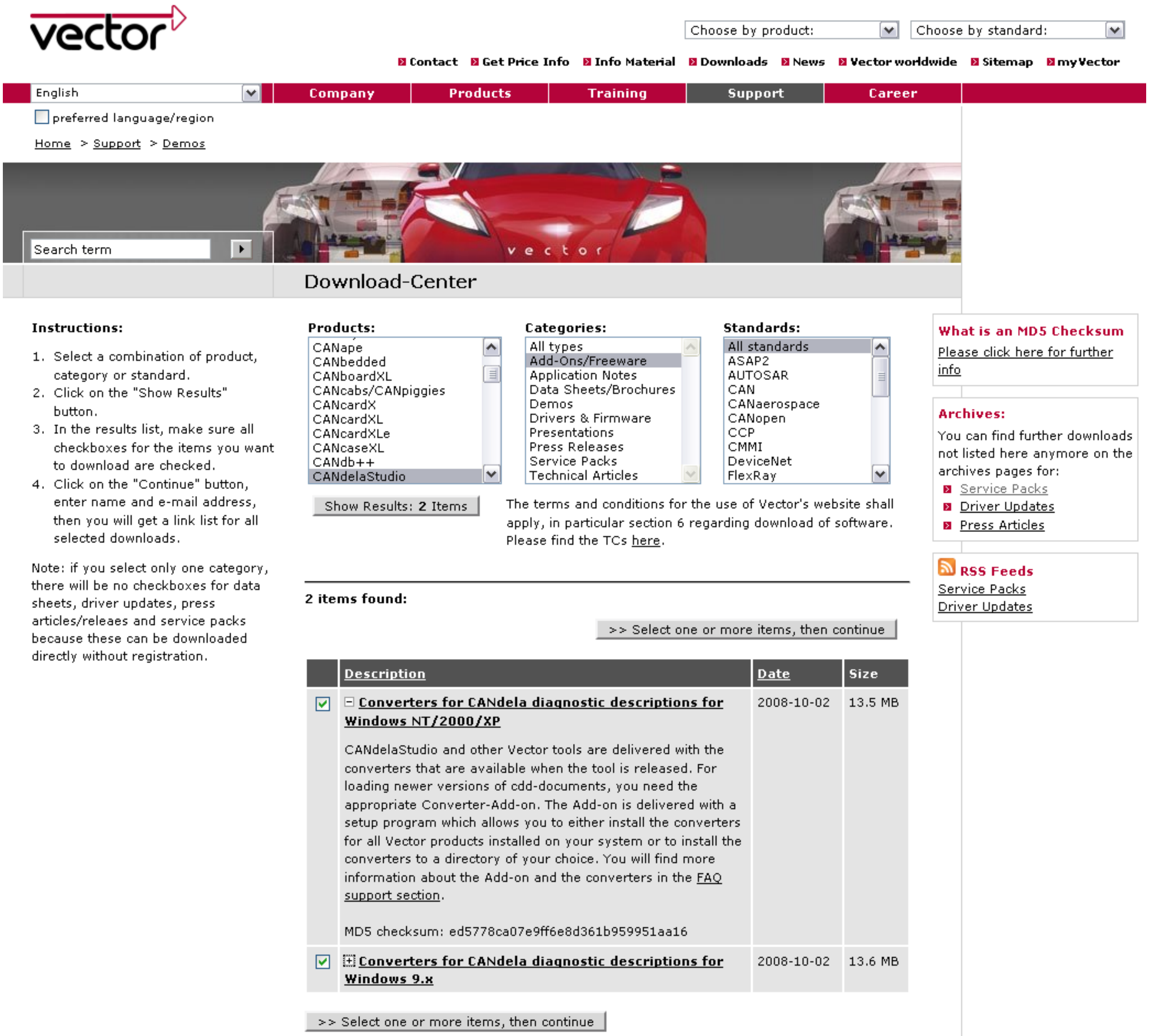
- Products: CANdela Studio
- Categories: Add-Ons/Freeware
- Standards: All Standards

Cross reference: See the following illustration.

Available for NT/2000/XP or Windows 9.x

The name for the Persistors download is:

- **Converters for CANdela diagnostic descriptions for Windows xxx**.



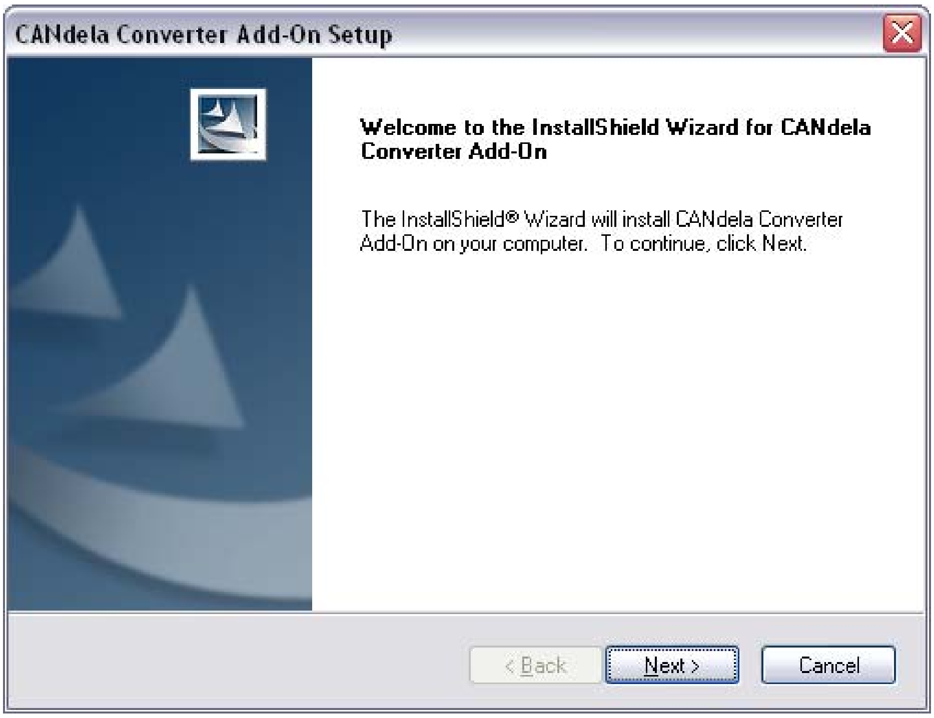
Download

Select on or more items from the list (☑) and click on **[>> Select one or more items, then continue]** to download the files after entering some administrative information.

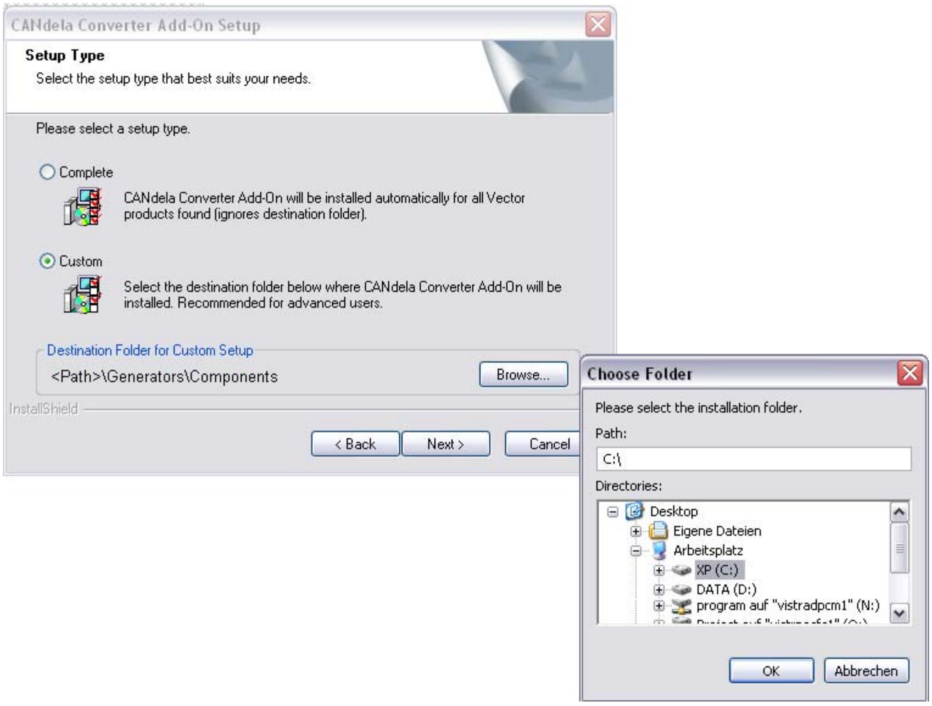
6.1.1 Update Persistors – Install current Version

Follow description step by step

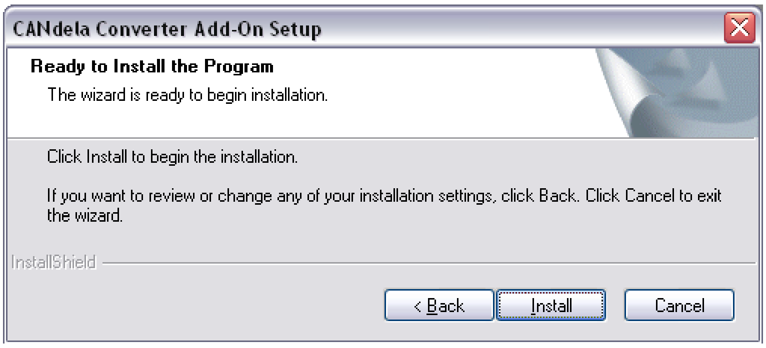
Start the downloaded file **SetupPersistorsXP.exe**.



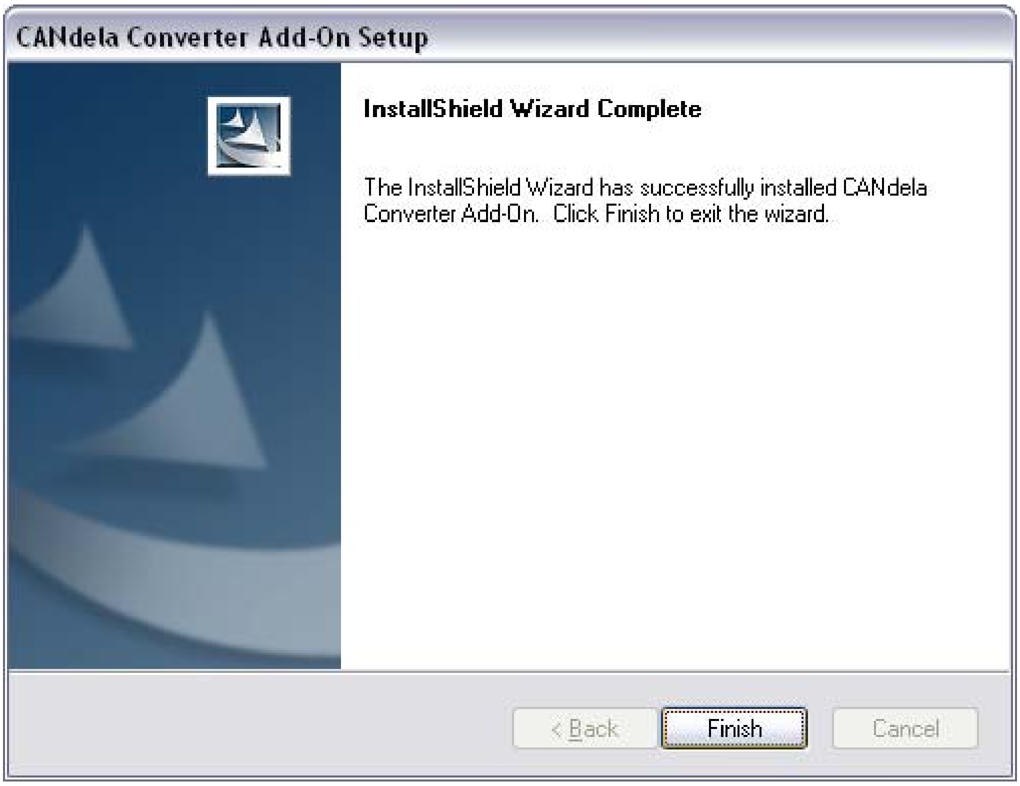
Click **[Next]**.



Select **Custom** and enter the path to the **…\Generators\Components** folder as **Destination Folder for Custom Setup** and click **[OK]**.



Click **[Install]** and the installation process will be started and then on **[Finish]** when ready.



Ready

Now the current Persistors are installed and your GENy is able to read the latest CDD file.

7 FAQs

In this chapter you find the following information:

7.1 Introduction

Find not search

You have a certain question? You just want to know how to do e.g. a certain setting without reading the whole document again?

Then go on reading the following list and use the links to get at the place in the document where your question will be answered.

This chapter will be extended continuously.

7.2 Frequently Asked Questions

FAQ: RingBuffer and the UDS SuppressPositiveResponseMessageIndicationBit (SPRMIB)

If the application wants to use the ring-buffer for a diagnostic service with a sub-function (usually service 0x19 "ReadDtcInformation") it shall consider the SPRMIB prior deciding to start the ring buffer. The reason for that is, once the ring-buffer response is activated this means to CANdesc that the application wants to send data. But if the SPRMIB=TRUE, there shall be no positive response on the communication bus. So in such cases the Application shall follow the sequence below:

```
if(pMsgContext->msgAddInfo.suppPosRes != 0)
{
  DescProcessingDone();/* just close the service processing
now. No response will be sent back*/
}
else
{
  DescRingBufferStart(); /* initate the ring-buffer response
transmission */
}
```

8 What’s new, what’s changed

In this chapter you find the following information:

8.1 Version 1.7

What’s new and what’s changed

This explains the changes within this document form the previous Version to the one mentioned in this headline.

8.1.1 What’s new

New chapter Persistors setup and update

There is a new chapter for additional information about Persistors setup and update at chapter additional information (see section Persistors on page 59).

8.1.2 What’s changed

New Layout

The Document has got a new template.

9 Address table

Vector Informatik GmbH	Vector Informatik GmbH Ingersheimer Str. 24 D-70499 Stuttgart Phone: +49 (711) 80670-0 Fax: +49 (711) 80670-111 mailto:info@de.vector.com http://www.vector-informatik.com/
Vector CANtech, Inc.	Vector CANtech, Inc. Suite 550 39500 Orchard Hill Place USA-Novi, Mi 48375 Phone: +1 (248) 449 9290 Fax: +1 (248) 449 9704 mailto:info@us.vector.com http://www.vector-cantech.com/
Vector France SAS	Vector France SAS 168, Boulevard Camélinat F-92240 Malakoff Phone: +33 (1) 4231 4000 Fax: +33 (1) 4231 4009 mailto:info@fr.vector.com http://www.vector-france.com/
Vector GB Ltd.	Vector GB Ltd. Rhodium Central Boulevard Blythe Valley Park Solihull, Birmingham West Midlands B90 8AS Phone: +44 121 50681-50 mailto:info@uk.vector.com http://www.vector-gb.co.uk

Vector Japan Co., Ltd.	Vector Japan Co., Ltd. Seafort Square Center Bld. 18F 2-3-12, Higashi-shinagawa, Shinagawa-ku J-140-0002 Tokyo Phone: +81 3 (5769) 7800 Fax: +81 3 (5769) 6975 mailto:info@jp.vector.com http://www.vector-japan.co.jp/
Vector Korea IT Inc.	Vector Korea IT Inc. Daerung Post Tower III, 508 Guro-dong, Guro-gu, 182-4 Seoul, 152-790 Republic of Korea Phone: +82(0)2 2028 0600 Fax: +82(0)2 2028 0604 mailto:info@kr.vector.com http://www.vector-korea.com/
VecScan AB	VecScan AB Theres Svenssons Gata 9 SE-417 55 Göteborg Phone: +46 (31) 76476-00 Fax: +46 (31) 76476-19 mailto:info@se.vector.com http://www.vecscan.com/

10 Glossar

Callback function	This is a function provided by an application. E.g. the CAN Driver calls a callback function to allow the application to control some action, to make decisions at runtime and to influence the work of the driver.
Diagnostics layer	Diagnostics services that are used in automotive applications have recently become standardized. As a result, basic requirements can be implemented by a software component for KWP2000/UDS.

11 Index

Get more Information!

Visit our Website for:

> News

> Products

> Demo Software

> Support

> Training Classes

> Addresses

www.vector-worldwide.com