

VDS Library Quick Start Guide

Version 1.3 | English

Imprint

Vector Informatik GmbH
Ingersheimer Straße 24
D-70499 Stuttgart

The information and data given in this user manual can be changed without prior notice. No part of this manual may be reproduced in any form or by any means without the written permission of the publisher, regardless of which method or which instruments, electronic or mechanical, are used. All technical information, drafts, etc. are liable to law of copyright protection.

© Copyright 2016, Vector Informatik GmbH. All rights reserved.

Table of Contents

1 Introduction	5
1.1 About this User Manual	6
1.1.1 Certification	6
1.1.2 Warranty	7
1.1.3 Registered Trademarks	7
2 Overview	8
2.1 General	9
2.2 Goal	9
2.3 Benefits	9
3 Quick Tour	10
3.1 Introduction	11
3.2 Include Vector.Diagnostics	11
3.3 Retrieve the Target ECU	11
3.4 Create a Request Object	11
3.5 Send a Request and Receive Responses	12
3.6 Evaluate the Result of the Sending	12
3.7 Access Parameters in a Request or Response	12
3.8 Get / Set Values of Diagnostic Parameters	13
3.9 Handle Parameter Iterations	13
3.10 Handle Multiplexer Parameters	14
3.11 Dispose Object Resources	14
3.12 Error Handling	15
3.13 Control of Tester Present Request Sending	15
3.14 Suppress Sending of Positive Response	15
3.15 Unlock ECU in Specified Security Level	16
4 Rules for Application Independent Scripts	17
4.1 Introduction	18
4.2 Restrictions	18
4.3 Reuse Scripts by Modularization	18
5 Example Script	19
5.1 Introduction	20

5.2 Complete Script Source Code (C#)	20
5.3 Using the Script	24
5.3.1 CANape	24
5.3.2 CANoe	24
5.3.3 Indigo	24
6 Frequently Asked Questions: Tips and Tricks	25
6.1 FAQ Overview	26

1 Introduction

In this chapter you find the following information:








1.1 About this User Manual	6
1.1.1 Certification	6
1.1.2 Warranty	7
1.1.3 Registered Trademarks	7

1.1 About this User Manual

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
Microsoft	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
	This symbol calls your attention to warnings.
	Here you can obtain supplemental information.
	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 9001:2008 certification. The ISO standard is a globally recognized standard.

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.

- > **Windows, Windows 7, Windows 8.1**
are trademarks of the Microsoft Corporation.

2 Overview

In this chapter you find the following information:

2.1 General	9
2.2 Goal	9
2.3 Benefits	9

2.1 General



API documentation: The detailed API documentation `VDSLibrary.chm` (online-help format) is contained in the installation. (Directories: CANoe: Help; CANape: Exec; Indigo/vFlash: Docs.)

2.2 Goal

Main goals

The **VDS Library (Vector Diagnostic Scripting Library)** provides a way to

- > automate the execution of diagnostic sequences
- > create, configure and send diagnostic request
- > receive and interpret diagnostic responses
- > share diagnostic procedures between **CANoe**, **CANape**, **Indigo** and **vFlash** (starting with CANoe 7.1. SP3, CANape 8.0, Indigo 1.0 SP2 and vFlash 2.7)

2.3 Benefits

.NET library

Basing the library on .NET provides the following benefits:

- > Any .NET programming environment may be used
- > Full development support is provided
- > Any .NET language may be used

Concept

The API was designed with these targets in mind:

- > Easy usage
- > Focus on application of diagnostic
- > Simple creation of small scripts and potential to build complex applications
- > Usage of standard .NET classes and self-made libraries is possible

Reuse

Scripts may be developed with one tester application (e.g. **CANoe**) and reused in every other application (e.g. **Indigo**, **CANape**). Please refer to Rules for Application Independent Scripts for details.

3 Quick Tour

In this chapter you find the following information:

3.1 Introduction	11
3.2 Include Vector.Diagnostics	11
3.3 Retrieve the Target ECU	11
3.4 Create a Request Object	11
3.5 Send a Request and Receive Responses	12
3.6 Evaluate the Result of the Sending	12
3.7 Access Parameters in a Request or Response	12
3.8 Get / Set Values of Diagnostic Parameters	13
3.9 Handle Parameter Iterations	13
3.10 Handle Multiplexer Parameters	14
3.11 Dispose Object Resources	14
3.12 Error Handling	15
3.13 Control of Tester Present Request Sending	15
3.14 Suppress Sending of Positive Response	15
3.15 Unlock ECU in Specified Security Level	16

3.1 Introduction

Function set at a glance

This section introduces the content of the library with small snippets of C# code. It is **not** a complete API description – please refer to the provided online help file for that.

3.2 Include Vector.Diagnostics

Concept

In order to use Vector Diagnostic Scripting, it is necessary to import the library into the .NET module.

```
// Import library definitions
using Vector.Diagnostics;
```

In complex applications it can be beneficial to use an alias for the namespace to prevent name collisions and make the source of a class more obvious.

```
// Import library definitions and assign an alias
using Diag = Vector.Diagnostics;
// To access the declarations the alias has to be put in front
Diag.Ecu ecuEngine = Diag.Application.GetEcu("Engine");
```

3.3 Retrieve the Target ECU

Concept

In order to communicate with an ECU, a script must retrieve an ECU object from the application.

Only diagnostic descriptions configured at the application may serve as target.

```
// Retrieve the ECU with the identifier "Engine"
Ecu ecuEngine = Application.GetEcu("Engine");
```

3.4 Create a Request Object

Concept

Requests are created at the ECU object by using either the qualifier of a diagnostic service (defined in the diagnostic description), or by providing a valid PDU as byte array.

```
// Create a request using the service qualifier
Request reqReadSWV =
    ecuEngine.CreateRequest("Software_Version_Read");
// Create a request using a (valid) PDU
Request reqRaw =
    ecuEngine.CreateRequest(new byte[] { 0x10, 0x03 } );
```

3.5 Send a Request and Receive Responses

Concept

A request is always in a state that allows sending it on the network configured in the application for its diagnostic description. The synchronous call waits until the request is processed completely, i.e. either all expected responses are received or a timeout / error occurred.

```
// Send request on the network and wait for responses/timeout
SendResult result = request.Send();
```

Hint

Alternatively you can send requests asynchronously by using the `SendAsynchronous()` method.

3.6 Evaluate the Result of the Sending

Concept

The send result value indicates whether sending the request succeeded and contains the responses received for the request (if the responses could be processed).

```
if (result.Status == SendStatus.CommunicationError)
{
    // Handle communication error
}
else if (result.Status == SendStatus.Ok)
{
    if (result.Response != null)
    {
        // Process response object
        Response response = result.Response;
    }
}
```

Note

For diagnostic services that may receive several responses (e.g. functional requests or certain fault memory requests), the property `Responses` is used to provide a list of response objects.

3.7 Access Parameters in a Request or Response

Concept

You can retrieve a parameter at its parent object via its qualifier. If a parameter cannot be found, null is returned. It is the responsibility of the diagnostic script to check the return value.

```
// Retrieve the parameter from the request with its qualifier
Parameter param1 = request.GetParameter("EngineSpeed");
if (param1 == null)
{
    // Error: parameter could not be found
}
```

Parameter hierarchy A parameter can hold child parameters that are accessible using the `Parameters` property. Either the qualifier of the sub-parameter may be used, or its index.

```
// Get the sub-parameter using its qualifier
Parameter subParam1 = param1.Parameters["MySubParam1"];
if (subParam1 != null)
{
    // Get the second parameter from the sub-parameter list
    Parameter subParam2 = param1.Parameters[1];
}
```

3.8 Get / Set Values of Diagnostic Parameters

Concept Parameters in message objects are initialized with the default values defined in the diagnostic description (when requests are created using a service qualifier), or by a PDU as raw byte array (e.g. response messages received from the ECU). Each parameter object has methods that give access to its value in different formats: a numeric parameter can be retrieved as a double value or as a formatted string, e.g.

```
double valueNumeric = param1.Value.ToDouble();
string valueSymbolic = param1.Value.ToString();
```

You can also set the value of a (request) parameter by calling the appropriate method. If setting the parameter is not possible, `false` is returned by the methods. It is the responsibility of the diagnostic script to check the return value.

```
// Try to set the parameter to a numeric value
if (!param1.Value.Set(27.3))
{
    // Could not set the parameter to the given value!
}
// Try to set the parameter to a symbolic value
if (param1.Value.Set("red"))
{
    // Setting the text table value succeeded
}
```

3.9 Handle Parameter Iterations

Concept Diagnostic services model lists of parameters using iterations. The number of iteration elements depends on its definition, e.g. an end-of-data iteration will fill up the complete message, while an end-marker iteration uses a specific value to indicate its end. The scripting library allows direct access to the iteration and its elements at the parameter that represents it.

```
// Get the list of DTCs in the response object
Parameter iterParam = response.Parameters["ListOfDTCs"];
if (iterParam != null)
{
    // The list of DTCs is available
    if (iterParam.Parameters.Count > 0)
    {
        // Since the list is not empty, get the first DTC
        Parameter dtc = iterParam.Parameters[0].Parameters["DTC"];
    }
}
```

Iterations in requests For a request parameter, it is possible to set the number of iterations by calling `SetIterationCount`. `False` is returned if this is not possible.

```
// Get the parameter that represents a list
Parameter iterParam = request.Parameters["ValueList"];
if (iterParam != null)
{
    // Modify the number of iteration elements
    if (iterParam.SetIterationCount(4))
    {
        // Set the value in the last iteration
        iterParam.Parameters[3].Value.Set(27);
    }
}
```

3.10 Handle Multiplexer Parameters

Concept



There is a special case where the value of one parameter determines the structure of one or more other parameters. For example, the selector parameter chooses between several parameter structures that become active depending on its value. If a selector parameter is set, the parameter structure of the request may change fundamentally, causing parameters retrieved earlier to become invalid. Therefore great caution has to be taken in this case; otherwise null parameters may be used.

```
// Get the multiplexer parameter and check its first parameter
Parameter mux = request.GetParameter("MUX");
Parameter firstParamInMux = mux.Parameters[0];
if (firstParamInMux.Qualifier == "Struct1_P1")
{
    // ...
}
// Get the selector parameter and change its value
Parameter selector = request.GetParameter("Selection");
selector.Value.Set(27);
// firstParamInMux is no longer valid!
if (mux.Parameters[0].Qualifier == "Struct27_P1")
{
    // The first parameter in the multiplexer is different now!
}
```

3.11 Dispose Object Resources

Concept

The resources that are held for messages (requests and responses) have to be cleared after the according objects won't be accessed any more. This has to be done by calling the `Dispose()` method of such objects. To assure that all (of the possibly multiple) responses of a service communication transaction are cleared, call the `Dispose()` method of the `SendResult` object.

```
Request request = ecuEngine.CreateRequest("...");
SendResult sendResult = request.Send();
// Perform actions on response
sendResult.Dispose(); // Free allocated response resources
request.Dispose(); // Free allocated request resources
```

Recommendation

As it is a common operation for managed languages (like C#) to explicitly dispose object resources, C# comes with an extra statement which is built-in in the language: 'using'.
With this keyword one can define a scope, outside of which an object will automatically be disposed.

```
using (Request request = ecuEngine.CreateRequest("..."))
{
    using (SendResult sendResult = request.Send())
    {
        // Perform actions on response
    } // Implicitly calls sendResult.Dispose()
} // Implicitly calls request.Dispose()
```

3.12 Error Handling

Concept

Error handling is performed according to these rules:

- > Return values are used wherever possible to indicate whether calling a method succeeded.
- > Exceptions are only used when it is necessary to indicate a fundamental problem and the application cannot continue executing the script.

Return values

Methods return true or false to indicate success or failure. If objects are returned, null indicates failure. The method `Request.Send` returns a more elaborate `SendResult` object that indicates the reason for failure in more detail.

3.13 Control of Tester Present Request Sending

Concept

It is possible to control the sending of Tester Present requests explicitly to keep the ECU in a specific diagnostic session, or let it drop to the default session after the sending has been deactivated.

```
// Toggle the sending of tester present requests
if (ecuEngine.IsTesterPresentActive())
    ecuEngine.ActivateTesterPresent(false);
else
    ecuEngine.ActivateTesterPresent(true);
```

3.14 Suppress Sending of Positive Response

Concept

The UDS protocol defines the possibility to instruct the ECU to suppress the sending of a positive response message. A flag has to be set in the request to activate this feature.

```
// Set the flag to tell the ECU to not send a positive response
request.SuppressPositiveResponse = true;
SendResult result = request.Send();
// Does not expect a response
```

3.15 Unlock ECU in Specified Security Level

Concept

To unlock an ECU in a specified level the tester has to perform the following steps:

- > Request the seed for the specified security level from the ECU
- > calculate the key based on the seed using the configured seed&key DLL in the client application (CANoe/CANape/Indigo)
- > send the key to the ECU and
- > evaluate the response of the send key service

The Vector Diagnostic Scripting library provides a convenient way to do all the steps in one call:

```
// Unlock ECU in security level 1
SecurityAccessResult result = ecu.Unlock(1);

// Check whether unlocking the ECU was successful
if (result == SecurityAccessResult.Success)
{
    // continue with unlocked ECU
}
```


4 Rules for Application Independent Scripts

In this chapter you find the following information:

4.1 Introduction	18
4.2 Restrictions	18
4.3 Reuse Scripts by Modularization	18

4.1 Introduction

Background

The applications providing the VDS library have different concepts, e.g.:

- > **CANoe** executes the scripts in a real-time context that allows no GUI elements, especially for automated ECU testing.
- > **CANape** focuses on the calibration of ECUs.
- > **Indigo** is a pure diagnostic tester that offers a use-case-driven GUI to apply diagnostics.
- > **vFlash** is a pure flash tool that offers custom actions.

Consequence

The applications offer very different use case libraries that cannot be present in every tool, therefore great care has to be taken to use only common functionality in diagnostic scripts intended to be portable.

4.2 Restrictions

Allowed libraries

Only libraries and methods that follow these restrictions may be used in portable scripts:

- > No parts of the threading API shall be used.
- > No GUI operations outside `Vector.Scripting.UI` shall be used.
- > Real-time processing must not be disturbed, i.e. blocking calls (e.g. for network or I/O operations, or complex XML parsing) have to be avoided.
- > Garbage collection of long duration should be avoided.

No Assemblies

The scripts can only be exchanged as source code files. Compiles assemblies are not supported for portable scripts.

4.3 Reuse Scripts by Modularization

Hint

A practical way to reuse diagnostic scripts is modularization:

- > Create one module containing only the diagnostic scripts.
- > Create application specific modules that import the diagnostic scripts module, and use the application specific features (e.g. test report generation in **CANoe**) only in those modules.

5 Example Script

In this chapter you find the following information:

5.1 Introduction	20
5.2 Complete Script Source Code (C#)	20
5.3 Using the Script	24
5.3.1 CANape	24
5.3.2 CANoe	24
5.3.3 Indigo	24

5.1 Introduction



Introduction: The following C# script will bring an ECU in the extended session, unlock it using the seed and key algorithm, and retrieve and print the variant coding values.



Info: Please have a look at the following sections for instructions on using the script in the Vector applications.

5.2 Complete Script Source Code (C#)

Initialization

Import the libraries needed in this script

```
using Diag = Vector.Diagnostics; // diagnostic functionality
using Vector.Tools; // output functions
```

Public class

Define a public class with default constructor to allow creation of an object of this type without additional information.

```
public class CodingReader
{
```

Tool function

The following function tries to create a request for the given ECU, and checks if this is possible. An error is printed to the write window and null is returned if it fails.

```
private Diag.Request CreateRequestWithCheck(Diag.Ecu ecu,
    string serviceQualifier)
{
    Diag.Request req = ecu.CreateRequest(serviceQualifier);
    if (req == null)
    {
        Output.WriteLine(
            "Error: Cannot create request for service "
            + serviceQualifier);
    }
    return req;
}
```

Tool function

This function sends a request and waits for the response. If the request cannot be sent or no response is received, null is returned, the received response otherwise.

```
private Diag.Response SendRequestWithCheck(
    Diag.Request request)
{
    // If no request is given return immediately
    if (request == null)
        return null;

    // Send the request and wait for the response
    Diag.SendResult result = request.Send();
    // Evaluate the result of the operation
    if (result.Status != Diag.SendStatus.Ok)
    {
        Output.WriteLine("Error: SendStatus = "
            + result.Status.ToString());
    }
}
```

```
// (Notice that this example assumes and requires that
// the ECU returns only one response!)
return result.Response;
}
```

Script function

A script function is any public method of the public class that does not expect arguments and returns void.

```
// Read the variant coding settings from the ECU
// The ECU has to be unlocked before the service may be
// executed, which can be done in the extended session only
public void ReadVariantCoding()
{
```

Access the ECU

Communication with an ECU can only be performed after the object representing it is retrieved from the application.

```
// Retrieve the ECU to communicate with
Diag.Ecu door = Diag.Application.GetEcu("Door");
if (door == null)
{
    Output.WriteLine("Error: Cannot access 'Door'!");
    return;
}
```

Extended session

The unlock algorithm may only be started in the “extended diagnostic session”, therefore the ECU is switched into this session. Since the positive response is of no interest here, the “suppress positive response” flag is set at the request.

```
// Enter the extended session, but do not expect a positive
// response
using (Diag.Request reqStart = CreateRequestWithCheck(door,
    "ExtendedDiagnosticSession_Start"))
{
    if (reqStart != null)
        reqStart.SuppressPositiveResponse = true;

    using (Diag.Response respStart =
        SendRequestWithCheck(reqStart))
    {
```

Session change failure

If the ECU sends a negative response, it is likely that it is in a non-default session already, so only a warning is printed and the script continues.

```
// Check if the ECU sent a negative response
if (respStart != null && !respStart.IsPositive)
{
    Output.WriteLine(
        "Warning: negative response received!");
}
}
```

Request seed value For the seed and key algorithm, the seed has to be requested from the ECU first. If this fails, the script cannot continue and will abort with an error.

```
// Request the seed for the unlocking procedure
using (Diag.Request reqSeed = CreateRequestWithCheck(door,
    "SeedLevel1_Request"))
{
    using (Diag.Response respSeed =
        SendRequestWithCheck(reqSeed))
    {
        if (respSeed == null || ! respSeed.IsPositive)
        {
            Output.WriteLine(
                "Error: No or neg. response received for request seed!");
            return;
        }
    }
}
```

Get parameters for seed and key The parameter representing the seed is retrieved from the response, and the parameter that holds the key is retrieved from a new request. If any of these parameters is not accessible, the script must abort with an error.

```
// Retrieve seed and key parameters
Diag.Parameter seedParam =
    respSeed.GetParameter("SecuritySeed");

using (Diag.Request reqKey =
    CreateRequestWithCheck(door, "KeyLevel1_Send"))
{
    Diag.Parameter keyParam =
        reqKey.GetParameter("SecurityKey");
    if (seedParam == null || keyParam == null)
    {
        Output.WriteLine(
            "Error: Seed or key parameter not accessible!");
        return;
    }
}
```

Compute key value The sample ECU uses a trivial algorithm to compute the key value from the seed value. It is set at the key parameter immediately for sending it in the request.

```
// The sample ECU uses a trivial algorithm to compute
// the key that can be coded easily
keyParam.Value.Set(
    0xFFFF - (seedParam.Value.ToInt32() & 0xFFFF));
```

Send key The request containing the computed key value is sent to the ECU. The response must be positive to indicate that the ECU has been unlocked successfully. Otherwise the script aborts with an error.

```
// Send key to ECU and check if pos. response was received
using (Diag.Response respKey = SendRequestWithCheck(reqKey))
{
    if (respKey == null || ! respKey.IsPositive)
    {
        Output.WriteLine("Error: Unlocking ECU failed!");
        return;
    }
}
}
```

Retrieve coding information

Once the ECU is unlocked, the coding information can be retrieved by sending the appropriate request.

```
// Request the coding information from the ECU
using (Diag.Request request = CreateRequestWithCheck(door,
    "Coding_Read"))
{
    using (Diag.Response response =
        SendRequestWithCheck(request))
    {
        if (response == null || !response.IsPositive)
        {
            Output.WriteLine(
                "Error: Could not retrieve coding!");
            return;
        }
    }
}
```

Output information

The parameters representing the variant coding information is output as text. Since a structure is used, the structure parameter is retrieved first, and the actual information is located in parameters that are children of the structure parameter.

```
// Output the information about the coding
Diag.Parameter codingStructure =
    response.GetParameter("Codingstring");
Output.WriteLine(
    "The ECU 'Door' is coded for '" +
    codingStructure.Parameters["Codingstring.CountryType"]
        .Value.ToString() +
    "' as a '" +
    codingStructure.Parameters["Codingstring.VehicleType"]
        .Value.ToString() +
    "' with special value '" +
    codingStructure.Parameters[
        "Codingstring.SpecialAdjustment"].Value.ToString()
    + "'");
}
```

Return to default session

In a final step, the ECU is put into the default session, locking it again.

```
// Return the ECU to default session
using (Diag.Request request = CreateRequestWithCheck(door,
    "DefaultSession_Start"))
{
    using (Diag.Response response =
        SendRequestWithCheck(request))
    {
        if (response == null || ! response.IsPositive)
        {
            Output.WriteLine(
                "Error: no or negative response received! ");
        }
    }
}
```

5.3 Using the Script

5.3.1 CANape

Introduction	The script can be loaded into the "UDS" demo delivered with every CANape installation.
Configuration	Open the "Tools" -> "Task manager..." menu dialog and go to the ".Net Scripts" tab. Press the "Insert" button and add the according file (e.g. named CodingReader.cs) containing the script. Then select the according method (e.g. "CodingReader::ReadVariantCoding") and press the "Select" button.
Execution	Within the Task Manager window this method now can be started using the "Start" button.

5.3.2 CANoe

Introduction	The script can be loaded into the "UDSsim" demo delivered with every CANoe installation.
Configuration	Open the "Automation Sequences" dialog (View Automation Sequences...), switch to tab ".NET Snippets" and "Add .NET snipping file..." (e.g. named CodingReader.cs) containing the script. After compilation, the .NET snippet "CodingReader::ReadVariantCoding" is available.
Execution	Open the "Automation Sequences" dialog (View Automation Sequences...), switch to tab ".NET Snippets" and execute the compiled script. Note: measurement must be running to execute scripts.

5.3.3 Indigo

Introduction	The script can be loaded into the Indigo demo – available e.g. via the Startup page.
Configuration	Open the "Script Manager" (Start Manage Scripts) and load the script with "Load Script ...". The script named CodingReader.cs is located in the <Examples> folder of the Indigo installation folder.
Execution	The script can be executed via the "Script Runner" window (Start Manage Windows ScriptRunner).

6 Frequently Asked Questions: Tips and Tricks

In this chapter you find the following information:

6.1 FAQ Overview	26
------------------------	----

6.1 FAQ Overview

Introduction

The following list is a collection of hints to make development of diagnostic scripts easier.

Question

FAQ: Where do I find the qualifiers of services and parameters?

Answer

There are several ways to access the qualifiers:

- > In CANdelaStudio (the “View” version is provided with every CANape/CANoe/Indigo, e.g.) you can open a CANdela diagnostic description and see the qualifier at the properties dialog of every object.
- > The ODXStudio (View Edition) (also provided with CANape/CANoe/Indigo) displays the content of ODX files and allows access to the “short names” that function as qualifiers.
- > The diagnostic console window (provided in CANoe, CANape and CANDito) can be configured to display the qualifiers of diagnostic services and parameters.
- > The CANape “Diagnostic Object Selection” dialog (available in the “Diagnostic objects...” context menu of the Function Editor) provides easy access to all those qualifiers.
- > The CANoe CAPL browser allows inserting service and parameter qualifiers into CAPL programs. You can open an empty CAPL program from CANoe and copy the output of the symbolic selection dialog (opens after pressing the right mouse button in an edit region) into your program.
- > The Indigo “Script Manager” window allows the generation of sample code for all services of selected ECUs. This sample code can be used for copy/paste of the services and the associated parameters.
- > The script recorder in Indigo allows to record scripts. This works in online (connected) mode as well as in offline mode. After storing the recorded script Indigo allows you to edit the script in Visual Studio with debugging support.

Question

FAQ: Is there a simple way to perform operations on every element of a parameter list or iteration?

Answer

A comfortable way to perform actions on every element of a parameter list or iteration is the standard C# foreach construct:

```
// Print the qualifiers of all child parameter
foreach (Parameter param in parentParam.Parameters)
{
    Output.WriteLine(param.Qualifier);
}
```



Get More Information

Visit our website for:

- > News
- > Products
- > Demo software
- > Support
- > Training classes
- > Addresses

www.vector.com