Using CANoe .NET API

Version 2.8

2018-02-12

Application Note AN-IND-1-011

Author	Vector Informatik GmbH
Restrictions	Public Document
Abstract	Describes CANoe .NET API usage details

Table of Contents

1.0 Document History

Version 2.8

> New chapter for using Exception Handling in .NET (chapter 7.6)

Version 2.7

> New document template and minor changes.

Version 2.6

> .NET4 support, test units, post compiler.

Version 2.5

> Glossary updated, minor changes.

Version 2.4

> Note about .NET framework version added (chapter 3.0)

Version 2.3

> Project templates for .NET test libraries that are independent from CANoe’s configurations

Version 2.2

> Document structure revised
> Glossary added
> Quick start instructions (chapter 0)
> .NET test libraries for XML and .NET test modules (chapter 4.3.4)
> Overview of reporting elements (chapter 5.2.2)
> Diagnostic tests (chapter 5.2.8)

2.0 Glossary

CANoe .NET API
A set of .NET assemblies (.dlls) providing access to CANoe’s functionality.

CAPL Library
A collection of CAPL test cases and test functions that can be used from your .NET program.

Snippet
A small piece of .NET program code that can be executed like a macro. Snippets can be changed and recompiled while measurement is running.

.NET Test Library
A .NET assembly containing further test cases, test functions or customer specific methods. A .NET Test Library is added as a component to a .NET or XML test module.

Runtime Values
Objects that represent signals, system variables and environment variables.

Type Library
A library that is generated by CANoe. It contains configuration specific data type definitions to comfortably access frames, signals, environment variables, system variables and CAPL library function from your .NET program.

Post compilation

After the .NET compilation a post compiler compiles the .NET assembly for a second time and adds code to control the test execution, error handling.

3.0 Introduction

The CANoe environment provides a .NET API to be used for simulation, test, and snippet programming. The CANoe .NET API is an Embedded Domain Specific Language extension that offers the possibility to use object-oriented programming languages, e.g. C# in the CANoe environment. .NET languages provide extended capabilities to structure, to reuse and to debug programs.

This document uses C# as recommended programming language. Nevertheless also other .NET languages like Visual Basic .NET can be used.

CANoe currently supports the .NET4 framework.

A .NET 2 version of the CANoe API is still provided for binary compatibility to your existing DLLs. If you have the sources it is recommended to switch to the .NET 4 API version, since the .NET 2 API is fixed to the functional range of CANoe 8.2 SP2. Further development will only support the .NET 4 API.

The .NET API can be used as an alternative to CAPL for test module or simulated node programming. It can also be used for programming so-called snippets, mainly used for simple stimulation or initialization purposes.

Discussed in this document is the .NET API concept and its usage in CANoe. It is assumed that the reader is familiar with the .NET framework and the application note **AN-IND-1-002 Testing with CANoe**.

This document does not primarily addresses test unit developers. However the chapters 5.0 and 7.0 are also relevant for test unit development.Quick Start

3.1.1 Prepare Visual Studio to be used as .NET Editor

1. Open the CANoe Options dialog (menu: Configuration | Options | External Programs | Tools)
2. On entry '.Net file Editor', click on '…'.
3. Select 'Exec32\Scripts\Edit_NET_Source_with_VS_20xx.vbs' from your CANoe installation folder (You need to adapt the file mask first to 'All files *.*'). To use the .NET4 framework at least VS2010 is required.
4. Confirm with 'OK'.

3.1.2 Add a .NET Test Module

A new .NET test module shall be added to an existing CANoe configuration.

5. Open the context menu with a right mouse click in the Test Setup. Select "Insert .NET Test Module".

6. Open the configuration dialog for the new test module. Enter a new test module name and a new source file name with a '.cs' extension.
7. Click on 'Edit'.

The test module skeleton and a solution is created and opened in Visual Studio. User added code is shown in bold and demonstrates some basic CANoe .NET API features.

```
using System;
using Vector.Tools;
using Vector.CANoe.Runtime;
using Vector.CANoe.Threading;
using Vector.Diagnostics;
using Vector.Scripting;
using Vector.Scripting.UI;
using Vector.CANoe.TFS;
using NetworkDB;

public class tester : TestModule
{

  public override void Main()
  {
    // test sequence definition:
    SimpleTest();
  }

  // Test cases need to be marked with an attribute:
  [TestCase("Simple Test")]
  public void SimpleTest()
  {
    Report.TestStep("Start engine:");
    // Setting bus signal SigStart to 1:
    NetworkDB.database1.SigStart.Value = 1;
    // Waiting 500ms for the SigEngine signal being 1:
    if (Execution.Wait<NetworkDB.database1.SigEngine>(1, 500) == 1)
      Report.TestStepPass("Engine is running.");
    else
      Report.TestStepFail("Engine is not running.");
  }
}
```

3.1.3 Create a CANoe configuration specific .NET Test Library

A new .NET Test Library shall be added to an XML test module. The library project shall contain all references to type libraries that are specific for the CANoe configuration.

8. Configure a temporary .NET test module in your CANoe configuration as described in 3.1.2. CANoe creates a solution with all references.
9. Remove the temporary .NET test module and continue implementation of the library functions with Visual Studio and the solution. The test library assembly must be located in the same folder as the XML test module.
10. With the 'Component' tab in the XML or .NET test module configuration dialog the test library assembly can be added. All test cases and functions are shown in the Test Automation Editor or are available in your .NET tester.

The .NET Test Library class does not need to derive from Test Module and to implement Main(). The example below shows the library with a test case and a test function:

```
using System;
using Vector.Tools;
using Vector.CANoe.Runtime;
using Vector.CANoe.Threading;
using Vector.Diagnostics;
using Vector.Scripting;
using Vector.Scripting.UI;
using Vector.CANoe.TFS;
using Vector.CANoe.VTS;
using NetworkDB;

public class NetLibrary
{
  // Test cases need to be marked with an attribute:
  [TestCase("Simple Test")]
  public void SimpleTest()
  {
    // test case as shown for a .NET test module, see above
  }

  [TestFunction("SetValue")]
  public void SetValue(int a)
  {
    NetworkDB.database1.SigStart.Value = a;
  }
}
```

Note:	this section does not apply to test units.

3.1.4 Create a .NET Test Library that is independent from a CANoe configuration

This chapter describes how to create a test library that is independent from the CANoe's configuration, i.e. it does not need references to configuration dependent type libraries. As an advantage such a library can be used in any CANoe configuration but some special handling is necessary when accessing configuration specific data, e.g. signals.

Visual Studio will offer a .NET test library template project when you copy

C:\...\CANwin Demos\Demo_Addon\VS_DotNetTestLibary_Template\CANoe DotNet Test Library.zip

to your Visual Studio CSharp Project Template folder, typically located here:

C:\Users\<username>\Documents\Visual Studio <vs_version>\Templates\ProjectTemplates\Visual C#

You can check the template folder in your Visual Studio Options dialog (menu: Projects and Solutions | General | User project templates location).

At next a new .NET test library can be created in Visual Studio with menu: File | New | Project | Visual C# and selecting 'CANoe DotNet Test Library'. Before confirming the dialog with 'OK' you should select a library name and the folder.

After a first compilation the assembly can be added to the XML tester components.

Note:	this section does not apply to test units.

3.1.5 Next Steps

> Change the test module to a structured Test Module that allows to enable individual test cases (chapter 5.2.1)
> Create test cases, test groups and define the execution sequence (chapter 5.2.1)
> Manipulate signals and environment variables (chapter 5.1.1)
> Verification of signals and environment variables (chapter 5.2.2), verdict generation and reporting (chapter 5.2.2)
> Setting up background checks (chapter 5.2.4)

4.0 Environment

4.1 CANoe .NET API Components

Two groups of assemblies build the .NET API and are needed for compilation of your .NET program:

> **CANoe programming interface**
These assemblies provide access to CANoe functionality. They are copied from the CANoe installation to the .NET solution folder in case of building the assembly from within Visual Studio. Nevertheless the solution will still refer to the assemblies of the CANoe installation.

> **Type libraries**
They are dynamically generated by CANoe and ensure type-safe and convenient access to database signals, messages etc. Type libraries are automatically updated when the underlying data is changed.
Database object will be represented by a class under the namespace NetworkDB. Environment variables are directly accessible under this namespace while message objects are always created in a subordinate namespace called Frames to prevent possible conflicts with signal classes. In case of name ambiguities of signals or messages, the namespaces of the generated libraries (e.g. qualification of signals --> DB.Node.Message.Signal) can be adjusted in the Options dialog in CANoe (menu: Configuration | Options | Programming | .NET). After changes are made in this dialog CANoe will regenerate the type libraries automatically. System variables are created under the user chosen namespaces.

When editing a .NET program (IDE started with CANoe) and no solution is found in the configuration folder CANoe creates a solution with all needed API references.

Note	CANoe supports generation but not modification of the VS solution/project file. If more components are added in the node configuration dialog or if a new database is added, these libraries must be added manually as references to the VS project.

In the following all type libraries and their functionality is described:

> **Vector.Tools**
Contains the shared API for the Vector tools, e.g. printing to the write window, measurement time access and timer management.
> **Vector.Tools.Internal**
Internal CANoe interface used by Vector.Tools
> **Vector.CANoe.Runtime**
Contains the interfaces and classes implemented by the CANoe runtime environment, e.g. event handler
> **Vector.CANoe.Runtime.Internal**
Internal CANoe interface used by Vector.CANoe.Runtime.
> **Vector.CANoe.TFS**
Contains the interfaces of the Test Feature Set used for test modules
> **Vector.CANoe.TFS.ITE**
Contains the interfaces of the Test Feature Set used for test units.
> **Vector.CANoe.TFS.Internal**
Internal CANoe interface used by Vector.CANoe.TFS.and Vector.CANoe.TFS.ITE
> **Vector.CANoe.Threading**
Contains the interfaces for e.g. wait commands, wait conditions and user input dialogs.
> **Vector.Diagnostics**
Contains the interfaces of the Diagnostic Feature Set.
> **Vector.Scripting.UI**
Contains further user dialogs for test nodes and .NET snippets.
> **Vector.CANoe.VTS**
Contains the API to access VT Systems.

> **Vector.CANoe.Sockets**
> Contains network access functionality.
> **<DB name>.dll**
> For all messages and signals and environment variables in the CAN database this assembly contains class definitions. The assembly is automatically generated and updated by CANoe.
> **<configuration name>.cfg_sysvars.dll**
> This assembly contains class definitions for all system variables used in the configuration file. The assembly is automatically generated by CANoe.
> **<system variable file>.dll**
> For each external system variable file CANoe creates a library file to access the system variables.
> **<program name>_CaplLibraries.dll**
> Contains the functions to access CAPL test cases and test functions from within .NET test modules

For a complete API description please check the CANoe online help.

Note1:	All .NET programs are executed in the runtime environment of CANoe and can slow down or even block the simulation.
Note2:	This document describes the .NET API available with CANoe version 8.2.
Note3:	For older versions some restrictions apply, please refer to the online help for detailed information

The .NET2 API additionally uses PostSharp.Public and PostSharp.Laos for post compilation purposes. They are not used in the .NET4 API anymore.

4.2 .NET Editor

Visual Studio .NET 2005 - 2013 can be used as IDE for .NET programs in CANoe. All IDEs are supported in CANoe by Visual Basic scripts that facilitate the automatic creation of solutions, project references, access to type libraries etc.

The open source environment SharpDevelop could be used as an alternative to Visual Studio but there is no script to support this IDE. The user must configure the environment manually, e.g. add all references etc.

To configure the .NET editor to be used in CANoe use the Options dialog (menu: Configuration | Options | External Programs | Tools | .NET file editor) and select the script (vbs file) matching your Visual Studio version. The available scripts for Visual Studio are delivered with the CANoe installation and located in the folder <CANoe installation path>\Exec32\Scripts.

Note:	It is important to configure the vbs script, and not the exe file of the editor, in CANoe.

4.3 .NET Programs in CANoe

Programs for network and test nodes can be specified in the node configuration dialog as .cs (C# source file), .dll (assembly) or .sln (a solution file).

A source file or a solution is used when the code should be visible and editable for the CANoe user.

Configuring a source file has the advantage that a solution with correct references is created for the module while this is not the case when you configure a solution.

One reason for using an assembly file might be to hide the information on how the module was programmed, or to restrain changes to the code. Note that the assembly must be rebuilt if the generated type libraries have changed. Please consider also the post compilation of assemblies (see also section 7.4).

A solution file should be used if the .NET program is subdivided in several source files or modules. It is recommended to let CANoe create the solution automatically. This is done by configuring a non-existing solution file name in the test module/node configuration dialog in CANoe and pressing the [Edit] button. The generated solution is opened with the chosen IDE, including all correct settings.

4.3.1 .NET Test Modules

Test modules should be added to the test environment in the Test Setup (see also **AN-IND-1-002 Testing with CANoe**). After adding a test module, you can right-click with mouse on the test module and choose Configuration… to specify the .NET file. CANoe will generate a skeleton of an unstructured test module when you edit it for the first time.

A .NET test module must be derived from the class `Vector.CANoe.TFS.TestModule or Vector.CANoe.TFS.StructuredTestModule`

Please find more info about test modules in chapter 5.2.

4.3.2 .NET Simulation Nodes

Simulated network nodes should be added to the Simulation Setup. After adding a simulated node, you can right-click with the mouse and choose Configuration… to configure the .NET file. CANoe will generate a skeleton file when you edit it for the first time.

A .NET simulated network node must inherit from the class `Vector.CANoe.Runtime.MeasurementScript.` Then you can handle measurement events like start/stop of measurement.

Note:	Types from Vector.CANoe.Threading, Vector.CANoe.TFS and Vector.Diagnostics cannot be used in simulated network nodes.

4.3.3 .NET Snippets

Snippets are small procedures that are often used for interactive test stimulation and/or initialization. Typically a snippet consists of one small function.

Snippets can be configured in the Macro and .NET Snippet Configuration dialog, under Configuration | Macro… Add a snippet by the button [Add] and configure the .NET source file here. Several snippets can be added in the list.

All public methods without parameters and without return value that are members of a public class in the .NET snippet file, are treated as snippets.

Macros and snippets can be run interactively. Snippets that are not running can be edited and compiled during measurement. Wait functions can be used in snippets and event handlers are active when the snippet is running.

Some restrictions apply to snippet programming (e.g. report methods cannot be used) – please see the CANoe online help for further information.

To learn more about the macros, and its configuration dialog, please refer to CANoe online help.

Note:	Types from Vector.CANoe.TFS cannot be used in snippets.

4.3.4 .NET Test Libraries

In contrast to normal .NET Test Libraries a .NET test library uses the CANoe .NET API and contains CANoe test cases respective test functions. Test cases and test functions from .NET test library can be invoked in XML and .NET test modules. A .NET test module - and not an XML Test module - can use normal .NET Test Libraries as well.

You can implement a test library that is independent from the CANoe configuration (see 3.1.4) or a library that contains references to the type libraries specific for a CANoe configuration, see 3.1.3.

A .NET test library consists of a public class which doesn't need to derive from any Vector API class and which does not need to implement the main() method. All public member functions with dedicated parameters and without return value that are marked with the TestCase or TestFunction are available in the test module. Event procedures in the .NET Test Library are only called during runtime of the .NET test case.

The resulting test library assembly can be added to an XML or .NET test module via the component tab of the node configuration dialog. The test library assembly must be located in the same folder as the test module.

When a test library itself references further assemblies these assemblies should be declared in the node configuration dialog. This way CANoe will be able to transfer all components when the simulation process is executed on a remote computer.

Note	All .NET Test Libraries under the Component tab in Test Module Configuration dialog will be added as references to the VS project of the test module, as long as they were configured *before* the creation of the test module project file.

It is important to differ between a .NET test module `(Main()` method is also implemented in .NET) and an XML test module with a .NET test case library.

In a.NET test module all variables are valid during the whole test. But if test cases from a library are used in an XML test module the test cases are independent from each other, i.e. each test case is instantiated once and variables are valid during test execution.

It should be noted that test cases implemented in a library may be called in an arbitrary sequence and should be treated according to this principle (initialization, usage, cleanup).

XML test module using .NET Test Libraries

.NET test cases are called within a `<nettestcase>` XML element:

```
<testmodule title="Central locking system test" version="1.0">
  <testgroup title="Test lock states of the car">
  <nettestcase name="Execute" title="Lock the car at great velocities"
       class="LockStateDependsOnlyOnVelocity" assembly="TestLibrary">
    <netparam type="float" name="ds">5.5</netparam>
      </nettestcase>
  </testgroup>
</testmodule>
```

.NET test functions are called with a `<nettestfunction>` XML element inside XML test cases. It is possible to call several different test functions within the same test case.

```
  <testcase title="My Test case Title" ident="">
    <nettestfunction name="MyTestFunction" title="Calling a NET test function"
       class="LockStateDependsOnlyOnVelocity" assembly="TestLibrary">
      <netparam type="int" name="myInt">10</netparam>
    </nettestfunction>
  </testcase>
```

name: specifies the name of the test case (mandatory)

title: used for reporting and visualization (optional)

class/assembly: used to identify the test cases (optional)

Parameter types can be numerical values, strings, signals, environment and system variables.

XML type	.NET type
int	UInt64, UInt32, UInt16, Int64, Int32, Int16
float	Double
string	String
signal	Vector.CANoe.Runtime.Signal
envvar	Vector.CANoe.Runtime.EnvironmentVariable
sysvar	Vector.CANoe.Runtime.SystemVariableBase

In contrast to the value types (int, float, string), the signal types are declared as Generics in the test case. Constraints must be applied to these Generics in order to guarantee type reliability. The specific types Signal, EnvironmentVariable or SystemVariableBase can be used for this.

The following definition is required if you want to use a test case for all three signal types similarly:

```
[TestCase]
public void SetSignal<T>(Double d) where T : class, IRuntimeValue
{
   RuntimeObject<T>.Value = d;
}
```

4.3.5 .NET Test Module and CAPL Test Cases / Test Functions

CAPL files can be assigned to a .NET test module with the test module configuration dialog. All CAPL test functions and test cases can be called from the .NET test module.

Not all CANoe features are available in the .NET API (this especially concerns bus system specifics) but with CAPL test libraries you can easily access the complete functionality.

CAPL libraries that are assigned to the .NET test module are structured like CAPL test modules, but they must not contain a `MainTest()` routine. Event procedures are only called during runtime of the .NET test module. All event procedures of the assigned CAPL libraries are considered except ‘on start’ and ‘on stop’.

CAPL test cases and test functions may use parameter types ‘char, char[], byte, int, long, float, double’. Results can be returned via system or environment variables.

A CAPL test case or test function can be called like any other .NET method. If they are used in structured test module they have to be wrapped by a .NET test case or a TestGroup method.

```
CaplTestCases.CaplTestLib.MyTestCase(“test”);
CaplTestFunctions.CaplTestLib.MyTestFunction(23.0);
```

Both have been defined in a CAPL library “CaplTestLib.can” and a reference for this library is configured in the .NET solution. The CAPL library looks like this:

```
testcase MyTestCase (char s[])
{
…
}
testfunction MyTestFunction (double d)
{
…
}
```

4.3.6 .NET Test units

Test units also use the CANoe .NET API. To create a test unit the Vector tool vTESTstudio is required. A detailed description of test units can be found in the vTESTstudio documentation.

Test units do not contain the Vector.CANoe.TFS.dll but the Vector.CANoe.TFS.ITE.dll.

Do not use Vector.CANoe.TFS in test units.

Do not use Vector.CANoe.TFS.ITE in test modules.

4.4 Additional Information and CANoe Examples

For a complete description of the API, including short code examples, please refer to: .NET API functional description in the online help of CANoe. More details to configuration settings etc. can also be found in the online help. A complete test module implemented in C# is available in the Demo directory of the CANoe installation (Start menu/Programs/CANoe/Demos/More demos/CANoe .CAN – Central Locking System .NET).

5.0 Programming with the CANoe .NET API

5.1 Common Features

5.1.1 Signals (bus signals, environment and system variables)

Signal types in .NET always mean bus signals, environment variables and system variables. These types are handled exactly in the same way. So each construct that works for one of these types does also work for the others.

Signal values can be accessed through the Value property, e.g. <signal name>`.Value`.

```
LockState.Value = 1;
```

In the signal-based view CAN, LIN and FlexRay bus signals are supported.

Note that setting the value usually doesn't change the value directly; in case of signals, the current value is only changed after a message which contains the signal has been transmitted on the bus and received again.

Environment variables can be handled in the same way, e.g.

```
MyIntegerEv.Value = 1;
```

In case the environment variable is a data array type you can easily read each byte with the index operator:

```
if (MyDataEv.Value[2] == 2) {…}
```

When assigning values you need to proceed as follows:

```
Byte[] tmp = MyDataEv.Value;
tmp[2] = 2;
MyDataEv.Value = tmp;
```

Easy support of resuing signal functionality can be realized by Generics.

```
public void SetSignal<T>(Double d) where T : class, IRuntimeValue
{
   RuntimeObject<T>.Value = d;
}
```

This function can be applied for all types of signals: `SetSignal<LockState>(1);`

It is also possible to dynamically create a signal object when its name is known. Here is an example how to create a system variable object (namespace = 'Tester' and variable name = 'Enabled') and to modify its value:

```
DynamicSystemVariable var = new DynamicSystemVariable("Tester", "Enabled");
var.Value = 1;
```

5.1.2 Messages

In the message-based view CAN messages are supported in a first step. For the CAN frames that are defined in databases, specific classes are generated in type libraries.

To construct a user-defined CAN message which is not defined in a database (e.g. dbc), the class CANFrame can be used as a super class. Signals are defined with the `[Signal]` attribute. Their offset and the least significant bit of the signal are as a bit number. Signal overlap etc will be checked at compile time.

```
class myMessage : CANFrame
{
  [Signal(LSB = 0, BitCount = 2)]
      public UInt32 mySignal;
}
myMessage msg = new myMessage(0x42,8);
msg.Send();
```

CAN messages can also be declared by creating instances of the `CANFrame` class and initializing the ID and DLC in the constructor. Message data can only be set in raw format, either byte-wise using the property `Bytes[]` or by a byte array which must be of size 8 bytes min:

```
byte[] data = new byte[] { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08 };
CANFrame msg = new CANFrame(0x500, 4); // ID=0x500, DLC=4
msg.SetRawData(data);
msg.Send();
```

To implement different send models CANoe is able to use so-called interaction layers realized by DLLs. OEMs often use specific interaction layers but there is also a generic Vector Interaction Layer available. If an interaction layer is used in the CANoe configuration, it is sufficient to set signal values without having to send messages explicitly – messages will be sent according to the correct send model by CANoe (e.g. cyclic messages will be sent automatically according to the required cycle time and sending mode).

Note that message objects in the configured databases are always created in a subordinate namespace called `Frames` to prevent possible conflicts with signal classes. Access to e.g. message VehicleMotion is accomplished through `NetworkDB.Frames.VehicleMotion.`

5.1.3 Timer

The attribute `[OnTimer]` can be used to accomplish a cyclic timer. The method will be executed cyclically at the defined time point (here 1s).

For test modules the timer is active when the test module is active and for simulation nodes it is active during measurement.

```
[OnTimer(1000))]
public void OnTimer1s() {
  Output.Writeline("Timer elapsed");
}
```

If a timer should be started and stopped at a certain point, timer objects can be used instead. Use the `Timer` class and its properties/methods, e.g. define a timer handler:

```
{
  Timer t = new Timer(new TimeSpan(0, 0, 0, 0, 250),TimerHandler);
  t.Start();
  …
  t.Stop();
}

public void TimerHandler(Object o, ElapsedEventArgs e)
{
 Report.TestCaseComment("Timer event");
}
```

5.1.4 Event Procedures

Event procedures of the main class are static, i.e. automatically activated and deactivated by CANoe. Event procedures in other classes can be dynamically activated and deactivated by using the class `Vector.CANoe.Runtime.DynamicHandlers`. Attributes are used to create event procedures.

For test modules the event handlers are active when the test module is active and for simulation nodes they are active during measurement.

A handler that reacts on value changes of the network signal LockState is defined like this:

```
[OnChange(typeof(LockState))]
public void OnSignalLockState() {
  double value = LockState.Value;
}
```

The .NET signal-oriented API only supports ‘on change’ handlers in contrast to CAPL that supports both ‘on change’ and ‘on update’ handlers. One reason is the performance aspect in .NET programs. Another reason is that the signal based tests are state-based.

In contrast to this message handlers (introduced with version 7.1) are event-based. To handle a message event for message WindowState use the `[OnCANFrame]` attribute. The CAN channel and message ID can also be specified in the attribute constructor.

```
[OnCANFrame]
public void onMessage(NetworkDB.Frames.WindowState frame) {
  double pos = frame.WindowPosition.Value;
}

[OnCANFrame(1, 500)]
public void Frame500Received(CANFrame frame) {…}
```

Use the attribute `[OnKey]` to react on key events. The defined method must have one `char` parameter.

```
[OnKey('+'))]
public void OnKeyPressed(char ch) {
  Output.WriteLine("+ was pressed");
}
```

To handle timer events the `[OnTimer]` attribute should be used. The method `TimerElapsed` below will be called every 500 ms:

```
[OnTimer(500))]
public void TimerElapsed() {
  msg.Send();
}
```

The module can also react on system variable changes:

```
[OnChange(typeof(MyNameSpace.MyInt))]
public void MyIntHandler()
{
   MyNameSpace.MyInt2.Value = MyNameSpace.MyInt.Value;
}
```

5.2 Test Features

5.2.1 Test Module Types

There are two different kinds of test modules:

- Unstructured test modules
  The sequence of test cases and groups will be displayed and completed during execution of the test module. It is not possible to enable individual test cases or groups.
- Structured test modules
  These modules have great advantages in comparison to unstructured modules because their structure is shown in the test execution dialog directly after compilation and they can be enabled or disabled before test execution.

Both module types are very similar and their format is described in the following chapters.

Unstructured test modules

An unstructured.NET test module must inherit from class `Vector.CANoe.TFS.TestModule`. It consists primarily of the `Main` method where the test control flow and the test cases are executed. Test groups can be used to organize test cases that belong together. Test cases are defined with the custom attribute `[TestCase]` and can be divided into test steps. The `Vector.CANoe.TFS.Report` class is used for reporting to the report file.

The `Main` method of the test module is the main program of the test module analog to a CAPL test module. It is required to override the inherited `Main` method of the `TestModule class`. The `main` method groups test cases and executes them sequentially:

```
public class MyTestModule : TestModule
{
  public override void Main()
  {
    TestGroupBegin("Id", "Title", "Description");
       TestCase1();
       if (TestModule.VerdictLastTestCase == Verdict.Passed)
       {
         TestCase2();
       }
       TestCase3();
    TestGroupEnd();
  }
}
[TestCase("Test case title")]
public void TestCase1()
{
    Report.TestStep("Description of the test step");
}
```

The `Main` method should only contain test flow control. The concrete tests should be done in the test cases. Complex flow logic can be implemented, e.g. letting a test case execute multiple times by calling it in a loop or prevent execution of a test case if a preceding test case returned failure.

In general test cases should be programmed such that they don't depend on each other, e.g. by not using global variables between test cases. It is recommended to start every test case from anew;

initialize parameters before test case start and – if needed – reset parameters after test case end. This ensures good programming practice and possible code re-use (which is one of the main motivations for the .NET API).

Event procedures are only active during test module run-time since test modules need not always be active during the whole measurement time (see also **AN-IND-1-002 Testing with CANoe**).

Structured test modules

A structured.NET test module must inherit from class `Vector.CANoe.TFS.StructuredTestModule`. The `StructuredMain` method of the test module is the main program and it is required to override the inherited `StructuredMain` method of the `StructuredTestModule class`.

It is required that dynamic test sequences (test sequences that may vary between two test runs) are encapsulated with methods that are marked with the [TestGroup] attribute. A dynamic test group will always be executed as a whole.

The example shows a DynamicTestGroup() and a normal test group with two test cases.

```
public class MyTestModule : StructuredTestModule
{
  public override void StructuredMain()
  {
    DynamicTestGroup();
    TestGroupBegin("Id", "Title", "Description");
       TestCase1();
       TestCase3();
    TestGroupEnd();  }
}
[TestCase("Test case title")]
public void TestCase1()
{
    Report.TestStep("Description of the test step");
}
…
[TestGroup("Dynamic Test case sequence"," depends on preceding test case results")]
public void DynamicTestGroup()
{
    TestCase1();
    if (TestModule.VerdictLastTestCase == Verdict.Passed)
    {
      TestCase2();
      // …
    }
}
```

The `StructuredMain` method should define the test sequence only. The concrete tests should be done in the test cases and test groups.

Event procedures are only active during test module execution (see also **AN-IND-1-002 Testing with CANoe**).

Note to structured test modules:

The test module structure analysis is done by tentatively running the test module in a special CANoe mode. During this execution the test case and dynamic test group bodies are skipped.

The special CANoe mode does not allow accessing runtime values and invoking methods from `Vector.CANoe.Runtime, Vector.Tools, Vector.Scripting.UI, Vector.CANoe.Threading` and most of the methods in `Vector.CANoe.TFS`. For this they cannot be called in the StructredMain() method.
It is not allowed to call CAPL test cases or test functions in the StructuredMain() method.

The execution sequence of the test cases in the module must be constant, e.g. a random execution of test cases is not allowed and may lead to unexpected enabling / disabling states of the test cases.

If you have a set of test cases that are randomly executed or that depend on the test verdicts from other test cases you should encapsulate them in a method that is marked with a [TestGroup] attribute. The body of such a method is also not executed during test module analysis and the group can only be enabled / disabled as a whole. An example for this can be found in the Central Locking Demo .NET of CANoe.

5.2.2 Reporting Commands

The report class allows detailed reporting of test results and test execution steps. The most important methods are listed here.

Informative:

> **Report.TestCaseComment(…)**	Some additional information that will be written to the report.
> **Report.TestStep(…)**	Separates the test execution into test steps.
> **TestCaseTitle(..)**	Modifies the test case title.

Verdict handling:

> **TestStepFail(…)**	The teststep verdict will be set to 'Fail' and it is propageted to the related test case and test module verdict.
> **TestStepPass(…)**	A passed test step with additional information is written to the report. The verdict is not changed.
> **TestStepWarning(…)**	A warning is written to the report.

A complete list can be found in the CANoe help or in the type libraries that are referenced in your test module solution.

5.2.3 Wait Points

Typically stimuli for the SUT are generated within the test case followed by waiting for the reactions. The sequential execution of a test module can be interrupted by so-called *wait points*. The principle of wait points is that they return flow control back to CANoe and do not resume processing until the specified event occurs (e.g. a signal value change, a message reception, or a timeout). For more details on the execution principle for a test module with wait points, please refer to **AN-IND-1-002 Testing with CANoe**.

There are wait points for simple timeouts, for dedicated values of bus or I/O signals, messages, for fulfillment of user-defined (complex) system condition, user interaction, etc. . The wait point is accomplished by using a `Wait` method of the `Execution` class:

Simple timeout:	`Execution.Wait(50);`
Bus signal:	`Execution.Wait<EngineRunning>(1);`
System variable:	`Execution.Wait<SystemUnderTest.OperationMode>(2);`
Environment variable:	`Execution.Wait<EnvDoorLocked>(EnvDoorLocked.Locked);`
Message:	`Execution.WaitForCANFrame(ref frame, 500);`

The signal-oriented API in .NET is state-oriented. This means that if a wait point for a signal element is used the wait condition is checked immediately and if the signal is already set (i.e. the condition is met), the wait point also resumes immediately.

In contrast to the signal-oriented approach in .NET message-oriented approach is event-based.

More wait points are contained in the API. Please check the online help for further details.

5.2.4 Checks

Checks in .NET test modules are used to monitor certain system conditions during a test sequence. There are three types of checks; Constraints, Conditions and additionally Observations.

Constraints are used to guarantee that the environment fulfills certain criteria and Conditions supervise the behavior of the tested system. Any violation of these checks leads to a failing test case and test module.

This is in contrast to Observations that cannot influence the verdict of a test module even though violations are always included in the test report.

Checks can be active during a complete test sequence, a complete test case, or they can be activated/deactivated using method calls.

The TFS provides several predefined checks

> Vector.CANoe.TFS.ValueCheck is used to monitor the value criteria of bus signals, environment and system variables
> Vector.CANoe.TFS.AbsoluteCycleTimeCheck, Vector.CANoe.TFS.RelativeCycleTimeCheck and Vector.CANoe.TFS.OccurenceTimeCheck check the timing of messages
> Vector.CANoe.TFS.DlcCheck monitors the message length indication

and the possibility to define custom checks:

> Vector.CANoe.TFS.UserCheck class is used to monitor customized system conditions.

The class `ValueCheck` represents a simple check for the value of a signal, environment variable or system variable. Here the value of the signal EngineRunning is checked:

```
[TestCase("Title", "Description")]
public void ObserveEngineState()
{
   ICheck engOn = new ValueCheck<EngineRunning>(CheckType.Observation, 1);
   engOn.Activate();
   Vector.CANoe.Threading.Execution.Wait(4000); // observation active
   engOn.Deactivate();
}
```

This is an example of checking the value of a bus signal. First the check for the signal value is created with the type Condition for the value ‘1’. Then the check is activated, and after a 4s wait the check is deactivated before the test case end.

A cycle time check for a message is done like this. The activation is done as in the previous example.

```
ICheck check = new AbsoluteCycleTimeCheck<NetworkDB.Frames.CyclicMsg>
                                             (CheckType.Condition, 80, 120);
```

If user-defined system conditions should be checked, a customized class can be defined (e.g. for DLC observation):

```
public class MyUserCheck : UserCheck
{
  public MyUserCheck(string title, string description)
    : base(title, description)
  {}
  [OnCANFrame(1, 0x64)]
  public void FrameReceived1(CANFrame frame)
  {
     if (frame.DLC != 1)
      ReportViolation("DLC check for frame 0x64 failed");
  }
}
```

The application of the check is as known from above:

```
ICheck check = new MyUserCheck("MyUser check", "Check DLC of message 0x64");
check.Activate(); // or: check.Activate("a new title");
// ...
check.Deactivate();
```

User checks can combine several runtime values as verification condition.

5.2.5 Criterions

Criterions can be used

- to wait until system conditions are reached
- to observe system conditions

The Criterion attribute can be used to define a handler to be used within checks or wait points

```
[Criterion]
[OnChange(typeof(AntiTheftSystemActive))]
[OnChange(typeof(LockState))]
bool AntiTheftSystemCriterion()
{
    if((EngineRunning.Value == 0) && (LockState.Value == 1))
        return (AntiTheftSystemActive.Value == 1);
    else
        return (AntiTheftSystemActive.Value == 0);
}
```

Apply the criterion above in a Check – the check fails, if the criterion handler returns false:

```
Check observeAntiTheftSystem = new Check(AntiTheftSystemCriterion);
observeAntiTheftSystem.Activate();
```

Apply the criterion above in as Wait Point condition – the wait point is resumed, if the criterion handler returns true:

```
Execution.Wait(AntiTheftSystemCriterion);
```

Note1:	The handler was immediately called on setup of the wait or the check to handle the start condition.
Note2:	Handlers that are marked with the Criterion attribute are only active while the check or wait condition is active.

The Criterion class can be used to combine several single criterions to a complex criterion, e.g. when several signal values have to be checked in parallel. This is similar to the `testJoinAuxEvent()` function in CAPL.

```
Criterion antiTheftSystem = new Criterion();
antiTheftSystem.AddMandatory(new ValueCriterion<AntiTheftSystemActive >(0));
antiTheftSystem.AddOneOf(new ValueCriterion<LockState>(0));
antiTheftSystem.AddOneOf(new ValueCriterion<EngineRunning>(1));
```

Again the combined criterion can be used I a check – the check is failed, if a mandatory condition is violated or if all optional conditions are violated:

```
Check observeAntiTheftSystem = new Check(antiTheftSystem);
observeAntiTheftSystem.Activate();
```

And also as a wait condition – the wait point is resumed, if if all mandatory conditions are fulfilled and if one optional condition is fulfilled:

```
Execution.Wait(antiTheftSystem, 1000);
```

5.2.6 User Dialogs

Original windows dialogs cannot be used in .NET test modules since they would disturb the CANoe real time behavior. Instead there are several predefined dialogs in `Vector.CANoe.Threading` and `Vector.Scripting.UI`. Here are some examples how to use them. Further dialogs are described in the online help of CANoe.

Tester Confirmation Dialog:

```
  int result = Execution.WaitForConfirmation("Please confirm!");
```

or

```
  int result = ConfirmationDialog.Show(("Please confirm!","Confirmation Dialog");
```

Value input with ranges:

```
   List<Range<uint>> l = new List<Range<uint>>();
   l.Add(new Range<uint>(1, 5));
   l.Add(new Range<uint>(10, 15));
   RangeCollection<uint> mRangesUint = new RangeCollection<uint>(l);
   result = DataEntryDialog.Show<uint>(
              "Please enter a value between 1-5 or 10-15! ",
              "ValueInput with range",
              "Two different ranges are defined.", ref value, mRangesUint);
```

5.2.7 Test Patterns

Test patterns are basic predefined test procedures or test flows that can be parameterized with values and that are executed in test cases. Typically the patterns are parameterized with input values and after a timeout the output (expected) values are checked. The usage of test patterns is one way of supporting the reuse concept.

In .NET test patterns are implemented as abstract classes, e.g. the class `StateChange`. These abstract classes must be instantiated as concrete classes with typed input and output (expected) vectors. The `StateChange` pattern implements the typical flow: stimulate, wait, evaluate – known also from XML test module patterns. The custom attributes `[Input]` and `[Expected]` are used to determine the input (stimulation) and output (evaluation) parameters.

First create the test pattern by deriving from the abstract class. Define the timeout, input and expected values.

```
public class CrashDetectionTest : StateChange
{
    public CrashDetectionTest() { Wait = 200; }
    [Input(typeof(NetworkDB.CrashDetected))]
    public double crashDetected = 1;
    [Expected(typeof(NetworkDB.LockState), Relation.Equal)]
    public double lockState = 0;
}
```

The created test pattern can now be used in a test case, e.g. by applying the `Execute()` method on it:

```
  [TestCase]
  public void LockStateDependsOnCrashDetection()
  {
      // Test Pattern - Crash Detection function test
      CrashDetectionTest crashTest = new CrashDetectionTest();
      crashTest.Execute();
      // Change test pattern input parameter and re-execute
      crashTest.crashDetected = 0;
      crashTest.Execute();
  }
```

User-defined test patterns can be defined with the [TestFunction] attribute:

```
[TestFunction]
private void InitSUT()
{
     // perform some actions
}
```

When executing the test function detailed reporting data is written to the test report.

5.2.8 Diagnostic Tests

Vector.Diagnostic is used to implement diagnostic request / response scenarios. A short example demonstrates this:

```
  [TestCase("Diagnostic Test")]
  public void TC_Diagnostic()
  {
    // select a diagnostic target ECU:
    Ecu myEcu = Application.GetEcu("ECU");
    // Create a request
    Request request = myEcu.CreateRequest("ECUOrgin_Read");
    // Send the request and wait for a response
    SendResult result = request.Send();
    // Check if the message transmission was successful
    if (result.Status == SendStatus.Ok)
    {
      // Get the response (Caution: the response might be null)
      Response response = result.Response;
      // Check if the ECU sent a positive response
      if (response.IsPositive)
      {
        Report.TestStepPass("Pos response received.");
         // Read and verify a parameter value:
    Parameter para = response.GetParameter("SerialNumber");
        if (para.Value.ToInt32 == 123)
          Report.TestStepPass("Serial number is OK");
        else
          ...
      } else {
        Report.TestStepFail("Negative Response received");
      }
    } else {
      Report.TestStepFail("Sending failed." + result.Status.ToString());
    }
  }
```

The service and parameter qualifier can be copied from the CANoe symbol explorer.

More background information can be found in ‘VDS_Library_QuickStart.pdf’ that is available in the CANoes start menu / Help / ‘Documentation Vector Diagnostic Scripting Library’.

6.0 Migrating a .NET Module that was created with CANoe < 7.6SP3

When a .NET module (network node, test node or test library) is created with CANoe < 7.6SP3 the solution contains absolute path references to the CANoe .NET API.

When you change to CANoe 7.6SP3 or newer these references and assemblies have to be updated with the actual ones that contain a link to the used CANoe installation:

- Remove the solution (*.sln) and the project (*.csproj) files for your module from your CANoe configuration folder. If the solution or project has user defined settings they have to be redone.
- In CANoe start editing the .NET module. A new solution with all needed references is created for the module. As a precondition you should configure the new CANoe version to use Visual Studio as editor, see also chapter 3.1.

6.1 Migrating from .NET2 API to .NET4 API

CANoe versions earlier than 8.2 SP3 provide the .NET API in .NET version 2. Starting with CANoe 8.2 SP3 the .NET API is provided in .NET version 4. Existing CANoe configurations stick to the .NET 2 version, but needs to be migrated to the .NET 4 version to benefit from further API development.

6.1.1 Migrating the CANoe configuration

The .NET4 API version is defined for each CANoe configuration in Options dialog (menu: Configuration | Options | Programming | .NET | .NET Runtime).

Because of internal changes the .NET2 assemblies are not binary compatible to the .NET4 API. So a recompile is required.

You cannot migrate to .NET 4 API if you are using compiled assemblies depending on an earlier CANoe version APIs of which you do not have the source code. In this case an appropriate error is reported in the CANoe Output window.

6.1.2 Migrating .NET Test Modules and Test Libraries

The .NET4 API uses the same API files as the .NET2 API, which are described in section 4.1. But the.NET4 version of the API is located in a different library folder: <CANoe program folder>/NETDev/**v40.** Existing test libraries reference assemblies of folder /NETDev.

For Test Modules, you typically reference one source code file. If you defined Visual Studio as editor as described in section 3.1.1, CANoe automatically creates a project file and a solution file for Visual Studio. You can safely delete the .sln and the .csproj file generated by CANoe. It will generate a new one referencing with the correct .NET runtime version setting referencing the correct CANoe API libraries, when you edit the test module.

If you changed the project file during development and would like to keep it, you can manually adjust it to the new CANoe .NET 4 API. The following sub chapter helps you to find the important modifications.

For Test Libraries, you can set up a new CANoe Test Library Project as described in section 3.1.4 to create a new project file which uses the .NET4 API. You can add your files to this template or merge the differences between your existing project and the template. The following sub chapter helps you to find the important modifications.

Manual project file modifications

Be sure to have following items contained in your migrated project file:

> .NET target framework version:
> <TargetFrameworkVersion>v4.0</TargetFrameworkVersion>

- Post compiler parameters:
  <PropertyGroup><PostCompileAdditionalParams>-R:$(CANoe_InstallDir)NETDev\v40 -R:$(CANoe_InstallDir)NETDev</PostCompileAdditionalParams></PropertyGroup>
- Post compiler invocation:
  <Import Project="$(CANoe_InstallDir)PostCompiler\Vector.PostCompiler.targets" />
- Typical .NET4 API reference block:
  <Reference Include="Vector.CANoe.TFS, Version=1.0.0.0, Culture=neutral, processorArchitecture=MSIL">
    <SpecificVersion>False</SpecificVersion>
    <HintPath>$(CANoe_InstallDir)NETDev**v40\**Vector.CANoe.TFS.dll</HintPath>
    <Private>**False**</Private>
  </Reference>
- Adjust these lines for each file contained in NETDev\v40 folder.
- Remove following references, which are not required:
  - ASAM.HILAPI.Interfaces.dll
  - ASAM.HILAPI.Implementation.dll
  - Vector.CANoe.ASAM.HILAPI.dll
  - PostSharp\PostSharp.Public.dll
  - PostSharp\PostSharp.Laos.dll
- Make sure to remove the corresponding using statements in the source code file.

7.0 Background Information (for Experts)

7.1 Real-time Performance

The precision and latency for .NET event handlers (e.g. `[OnChange]`) is the same as for CAPL event handlers. There is no additional latency due to .NET but it should be noted that real-time performance always depends on the PC and HW configuration. If real-time performance is an issue CANoe can be configured to execute the RuntimeKernel on a dedicated CANoe RT PC (for more information, please see the CANoe online help).

The .NET Framework's garbage collector manages the allocation and release of memory for the application automatically. This process may lead to unexpected latency times of memory requests. Therefore .NET code is executed in a special way in CANoe to prevent unexpected latency times:

The CANoe real-time kernel (RuntimeKernel.exe) is executed in a separate process whenever .NET code is used.

Please follow these general recommendations if real-time performance is an issue for the application:

- Avoid extensive memory requests during a simulation
- Avoid long running algorithms that are not interrupted
- Do not perform I/O operations during simulation
- Do not use GUI elements, e.g. message windows
- Refer to relevant literature on MSDN web pages regarding garbage collection

7.2 Concurrency

Although there can be several test modules running in parallel, these modules are not executed in separate threads. This means you don't have to be concerned with synchronization mechanisms. While a test module is waiting for an event, other test modules or CAPL code are carried out. This is a form of cooperative multitasking; for this reason you should make sure not to program long running operations in a test module without using intermediate Wait calls.

Because of the CANoe internal concurrency architecture, you are not permitted to use synchronization primitives (such as locks) or start additional threads in a .NET program.

If you need to execute potentially long lasting operations you can use the CANoe `WaitForTask` function. It is also possible to use the `System.Thread` class. In both cases only a very limited number of API-functions are available in background threads:

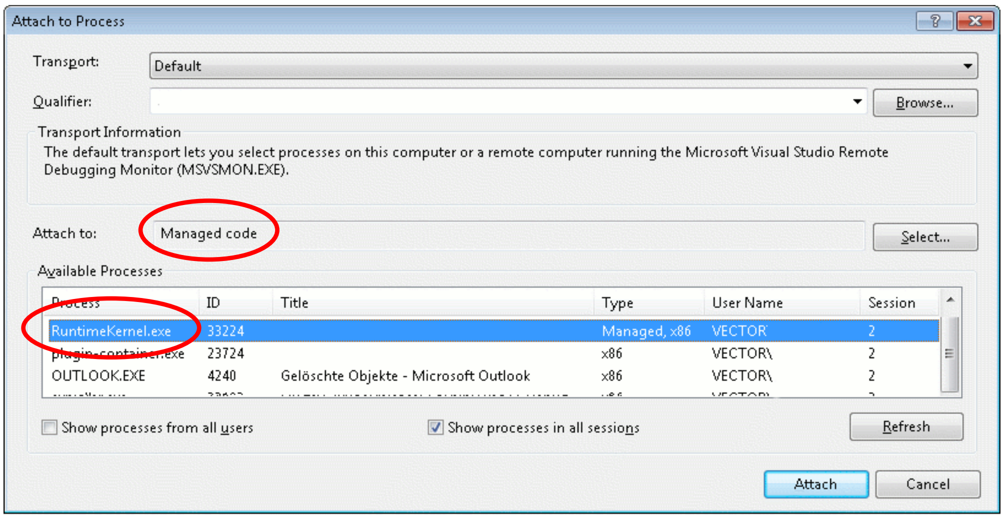
- Writing system variables (not reading!)
  Hint: Test system variables in test units are not supported
- Writing to log file
- Writing messages to output window

All other API function will throw an Exception.

Note:	If class `System.Thread` is used, the user code is responsible for the exception handling. In contrast to chapter 7.6, a catch on the base class `System.Exception` must be used here.

7.3 Debugging .NET Programs

All .NET programs are executed in the CANoe real time task 'RuntimeKernel.exe'. For debugging the .NET program the debugger has to be attached to the RuntimeKernel.exe with 'managed code type' option. The RuntimelKernel.exe will be started when CANoe measurement is started for the first time.



Attach to Process

Transport: Default

Qualifier:

Transport Information

The default transport lets you select processes on this computer or a remote computer running the Microsoft Visual Studio Remote Debugging Monitor (MSVSMON.EXE).

Attach to: Managed code

Available Processes

Process	ID	Title	Type	User Name	Session
RuntimeKernel.exe	33224		Managed, x86	VECTOR	2
plugin-container.exe	23724		x86	VECTOR\	2
OUTLOOK.EXE	4240	Gelöschte Objekte - Microsoft Outlook	x86	VECTOR\	2

☐ Show processes from all users

☑ Show processes in all sessions

When the .NET module has been built with CANoe there is normally no debugging information available and you should first rebuild the assembly with enabled debug mode in the IDE or you can enable .Net Debugging in CANoe in Options dialog (menu: Configuration | Options | Programming | .NET Debugging | .NET Debugger)

In case you debug your module in CANoe's simulation mode it is useful to enable 'windows timer' in Options dialog (menu: Configuration | Options | Measurement | General | Simulation | Use Windows timer).

For detailed information on how the debugger should be attached, please refer to the Visual Studio documentation.

7.4 Post-compilation

CANoe uses a PostCompilation step to add CANoe specific behavior to the .NET assemblies. Source code containing the Attributes TestCase, TestFunction, Signal,,KeyboardHandler, TimerHandler, TimerHandler etc. are enriched with additional code sequences used for e.g. test control, error handling, reporting features.

The post compilation of the .NET assemblies is done by CANoe for all kinds of .NET modules. The post compilation of the .NET assemblies is usually done in the Visual Studio build process, if the project was created by CANoe. This is the recommended way of post compliation.

However CANoe checks .NET modules to ensure they are post compiled before using them. So a post compilation step is automatically added in CANoe compile, if required.

To manually check if your .NET assembly is post compiled check an included resource:

```
        public resource Vector.PostCompiler.Interface.IsPostCompiled
```

7.5 Troubleshooting

7.5.1 Debug Information not visible (breakpoints not hit)

Check that the debugger was attached to the RuntimeKernel.exe in managed mode and not the CANoe process. See chapter 7.3 for more information regarding this topic.

7.5.2 Intellisense not active in Visual Studio

Check that the vbs script (e.g. Edit_NET_Source_with_VS_2005.vbs) and not the exe file of Visual Studio, is configured in CANoe Options dialog (menu: Configuration | Options | External Programs | Tools | .NET file editor).

7.5.3 Warning in Write window using a Test Case Library

The following warning can occur in write window of CANoe when developing a test case library using `Vector.CANoe.TFS.Report.TestStep`:

```
'Test Test module '<Test module name>': TestStep ignored. There is no test case
running!'
```

This warning can occur if the test case library was not post-compiled (see section 7.4).

7.5.4 Visual Studio cannot build because annother process holds the .NET assembly

Sometimes the RuntimeKernel.exe holds the assemblies after the measurement has been ended.

This problem can occur if the .NET solution/source file was created manually (i.e. not automatically by CANoe). If the file was created manually the problem can be fixed by adding the following in Pre-build event command-line in Visual Studio:

```
"$(CANoe_InstallDir)Scripts\ReleaseDotNetAssemblies.vbs" "$(SolutionPath)"
```

A new solution with correct settings can be created by deleting the old solution and editing the node. CANoe creates a solution with all needed API references and commands.

7.5.5 You encounter build errors due to wrong API versions

If you have more than one CANoe version installed, make sure you have registered the CANoe version you are currently using.

If in doubt, use the RegisterComponents.exe (as Administrator) in CANoe Exec32 folder.

You can check the windows environment variable `CANoe_InstallDir`. It must point to the CANoe installation you're working with.

7.6 Exception Handling

If you need some exception handling inside your .NET source code please don´t use the base class `System.Exception` in the catch Handler (except together with `System.Thread` – see chapter 7.2). Either use a predefined exception class or define your own exception class that inherits from the `System.Exception` class.

Example:

```
[TestClass]
public class TestCaseLibrary
{
  [Export] [TestCase] [BreakOnFail(true)]
  public static void TestWithOwnExceptionHandling()
  {
    try
    {
      //Do something you need exception handling
      …
    }
    catch(MyConcreteException)
    {
      //Do something inside your catch-block
      …
    }
  }

  //Override the "System.Exception"-class
  private class MyConcreteException: System.Exception {}

}
```

8.0 Additional Resources

VECTOR APPLICATION NOTES: HTTP://WWW.VECTOR.COM/APPNOTES/

AN-IND-1-002 TESTING WITH CANOE

9.0 Contacts

For a full list with all Vector locations and addresses worldwide, please visit http://vector.com/contact/.