Diagnostics with CAPL

2023-07-27
Support Note SN-IND-1-040

Author(s) Ernst, Oliver; Schwarz, Dirk
Restrictions Public Document

Table of Contents

1 About this Support Note

In the table below you will find the icon conventions used throughout the Support Note.

Symbol	Utilization
	This icon indicates notes and tips that facilitate your work.
	This icon gives you step-by-step instructions.
	This icon indicates examples.

2 Overview

This Support Note explains how to use the diagnostic functions provided by the CAPL programming language in CANoe and in CANalyzer. Only diagnostics on CAN is covered, but aside from the bus specific aspects, diagnostics on FlexRay, LIN, K-Line and DoIP is quite similar.

This Support Note is intended for CANoe versions starting from 9.0 SP3 and higher.
There is a separate Support Note for older CANoe versions. To receive it, please contact the Vector Support (contact information in chapter 18).
All screenshots in this document are taken from CANoe 14.

3 Configuring the diagnostic components of CANoe/CANalyzer

If you want to create diagnostic tests or if you want to access the diagnostic data of an ECU with CANoe/CANalyzer, you have to add a diagnostic description first. There are different types of descriptions available:

- **CDD (CANdela Diagnostic Description)**
  CDD files are created in the Vector tool CANdelaStudio.

- **ODX (Open Diagnostic Data Exchange)**
  Since the diagnostic data can be divided into several ODX files, the description is usually provided as a single PDX (packed ODX) file. A PDX can contain the diagnostic data for more than one ECU. Therefore, you must select the ECU in this case.

- **MDX (Multiplex Diagnostic Data Exchange)**
  This is an OEM-specific format.

If none of these concrete descriptions is available, CANoe/CANalyzer offers two alternatives:

- **Standard Diagnostic Description**
  Predefined CDD files delivered with CANoe/CANalyzer which contain only services defined in the ISO standards. The CDD files cannot be customized.

- **Basic Diagnostic Description**
  This description must be created by the user inside CANoe/CANalyzer using the Basic Diagnostics Editor.

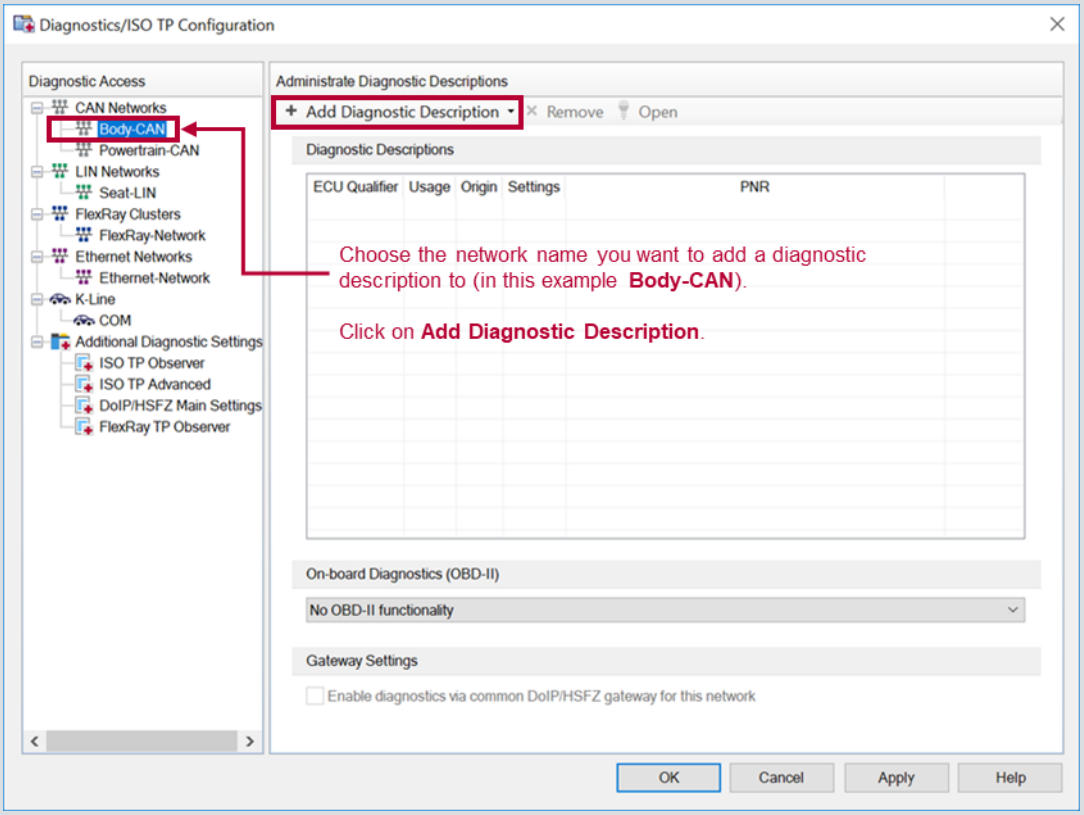
They have only limited functionality (e.g. Basic Diagnostic Descriptions do not contain a fault memory model, session model or security access).

3.1 How to add a diagnostic description in CANoe/CANalyzer

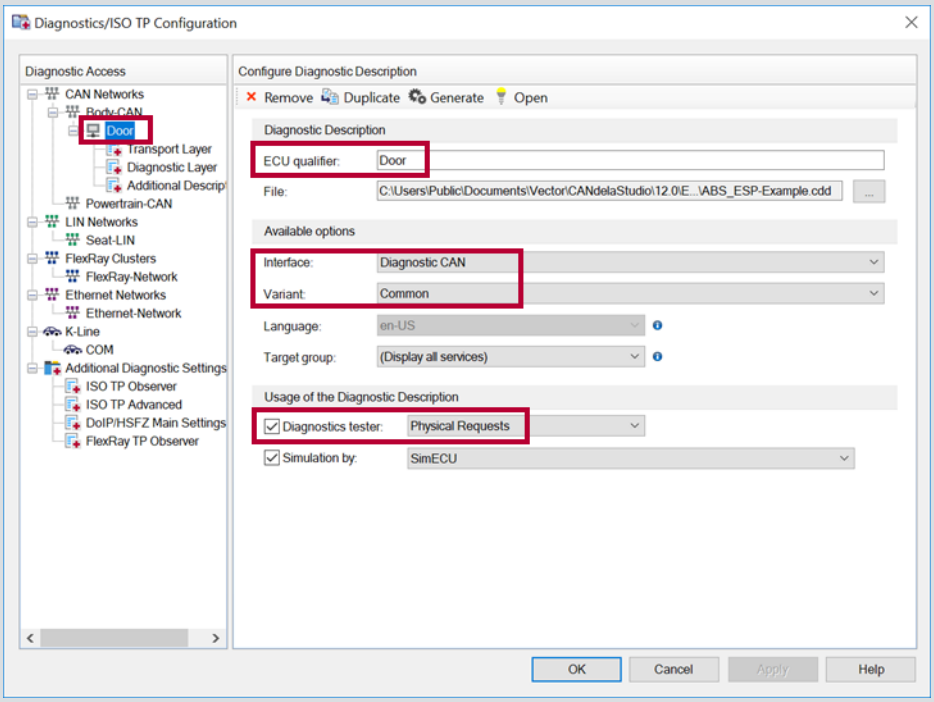
> In the **Diagnostics & XCP** ribbon, click on **Diagnostics/ISO-TP:**

The **Diagnostics/ISO TP Configuration** window will appear.

> Choose the targeted network name and click on the button **Add Diagnostic Description**. Select one diagnostic description type in the appearing drop-down menu and select the file (except for basic diagnostics where the Basic Diagnostic Editor will open after closing this dialog box).



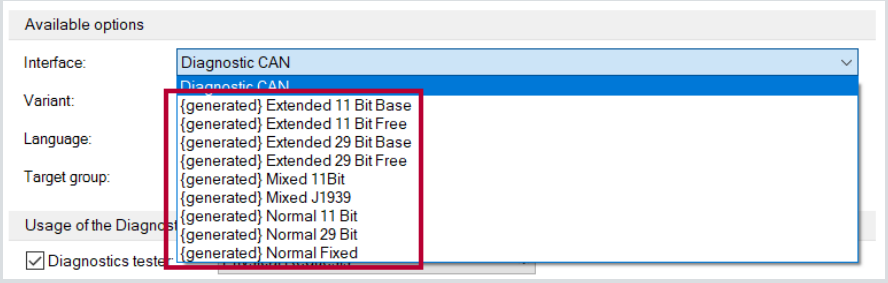
After the selection of the diagnostic description, the diagnostic configuration for the chosen network is available:



The configuration`s branch is named after the ECU qualifier (here **Door**).
The most important settings are:

> **ECU qualifier**
The qualifier is a unique identifier for this diagnostic description.
It is also used in CAPL to address diagnostic requests (see chapter 5).

> **Interface**
The interface is a set of communication parameters to access the ECU.
In case the diagnostic description does not contain a valid or suitable set of these parameters, you can use some default interfaces (prefixed with {generated}) for 11-bit and 29-bit addressing, for normal fixed addressing (for J1939), extended addressing, etc. see screenshot below:



> **Variant**
The variant determines which services are available for diagnostics.
If the diagnostic description contains more than one variant, please choose the desired one here.

> **Diagnostics tester**
Please determine which type of requests CANoe/CANalyzer as diagnostics tester should use:

> **Physical Requests**
The tester will send physical requests in order to access only one ECU on the network. This is the typical way of diagnostic communication.

> **Functional Group Requests**
The tester will send functional requests in order to access all ECUs on the network.

3.2 Property Pages

The configuration's branch has three property pages:

> Transport Layer

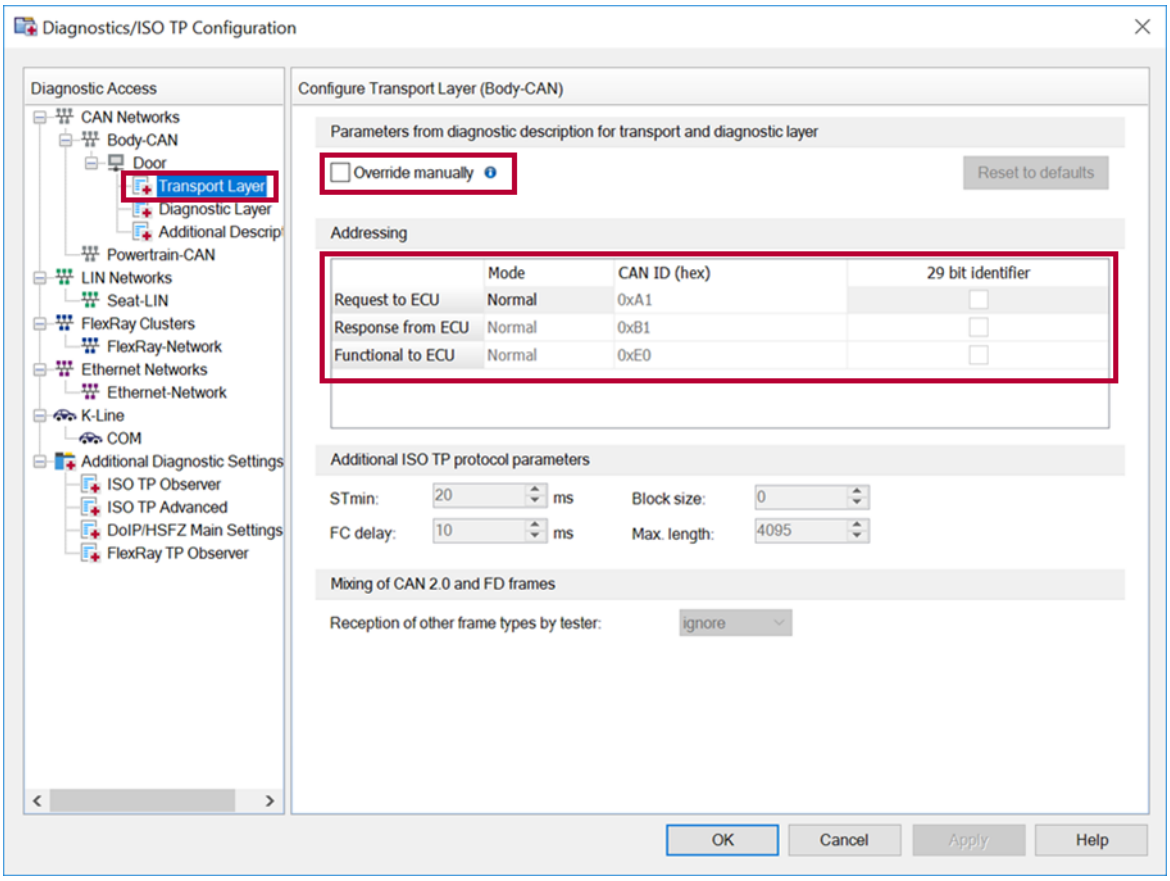
> Diagnostic Layer

> Additional Descriptions

The most important settings of these three pages are explained below.

3.2.1 Transport Layer

The settings on this page depend on the bus type. For CAN it looks like this:

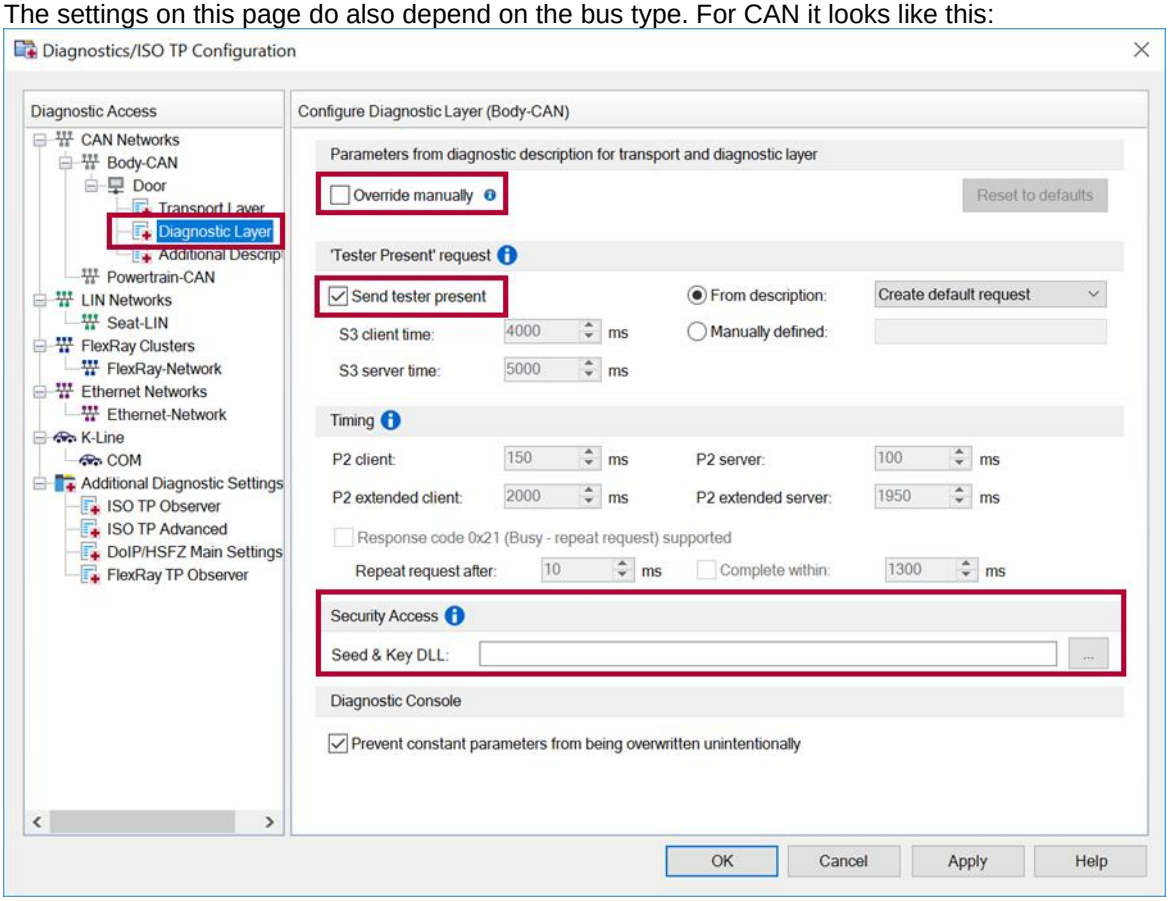


> **Override manually**
You can decide whether to use the communication parameters of the selected interface or to overwrite them and use different values. Depending on the addressing method (physical or functional requests) some or all of these parameters on this page are editable. If you have selected one of the default interfaces (prefixed with {generated}), the override manually setting is automatically checked as you are required to make some settings (e.g. like CAN ID).

> **Addressing**
Here you can set the CAN IDs used for functional or physical requests and responses when **Override manually** is enabled. It is also possible to activate 29-bit identifier via the check box.

3.2.2 Diagnostic Layer

The settings on this page do also depend on the bus type. For CAN it looks like this:



> **Override manually**
You can decide whether to use the S3 and P2 timings of the selected interface or to overwrite them and use different values.

If you have selected one of the default interfaces (prefixed with {generated}) the override manually setting is automatically checked.

> **Send tester present**
With this checkbox you can enable or disable if Tester Present should be sent cyclically to the ECU. Tester Present is only sent after the measurement has been started and once a diagnostic request has been sent.

> **Security access**
Here you can include a Seed & Key DLL which provides the security algorithm for unlocking the ECU.

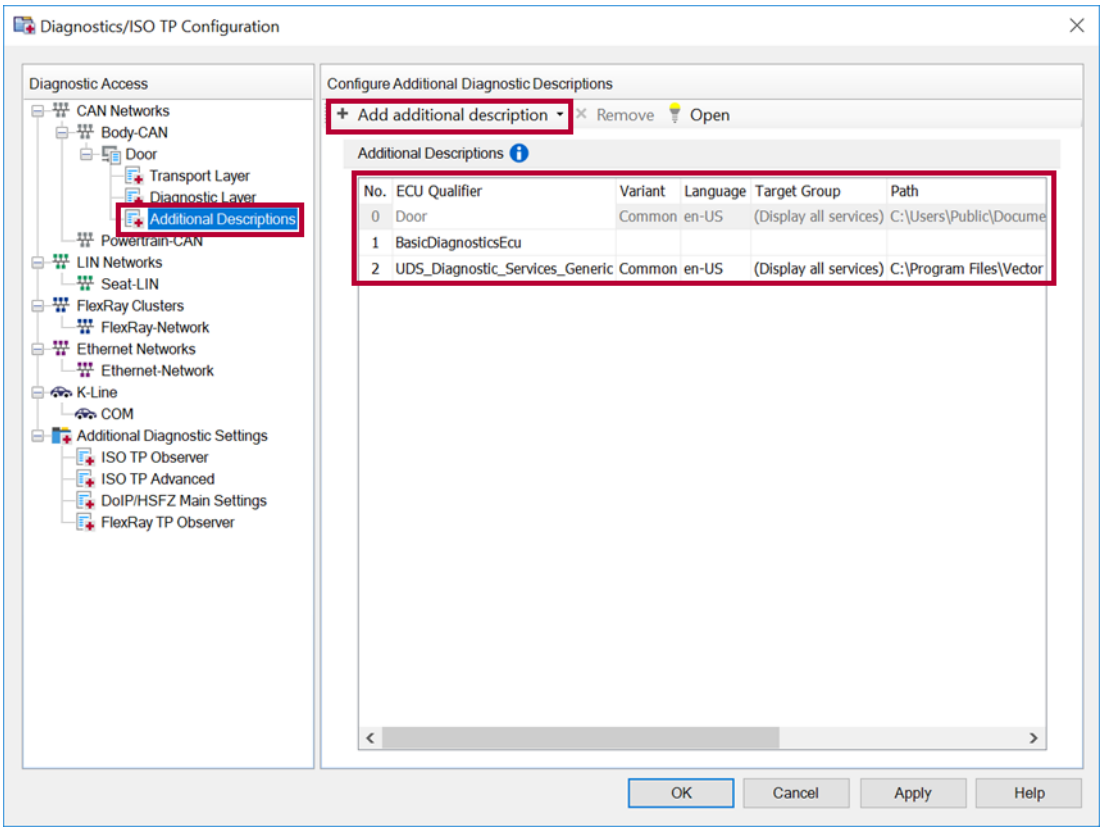
Two templates for creating such a DLL with Visual Studio (including the project file) can be found in the following directory:
`C:\Users\Public\Documents\Vector\CANoe\Sample Configurations [CANoe-Version]\CAN\Diagnostics\UDSSystem\SecurityAccess`

For information about security access with CAPL please refer to chapter 11.

3.2.3 Additional descriptions

If you want to use a service that is not defined in the added diagnostic description (referred to as master description) you can use additional descriptions to extent the Master Description. This is especially useful if the master description cannot or shall not be changed.

This property page is bus independent and therefore always looks like this:



> **Add additional description**
Like for the master description you can choose between the following description types:

Diagnostic Description (CDD, ODX/PDX, MDX)
Standard Diagnostic Description
Basic Diagnostic Description

It is also possible to add more than one additional description (as depicted in the screenshot above).

> **Additional descriptions table**
The table shows the master description (in grey) and the added additional descriptions.
The settings (e.g. **ECU Qualifier** or **Variant**) for the additional descriptions can be changed here.

You can drag the lines with the descriptions up or down to set the search order within the descriptions. The top description is taken first and the description on the bottom is taken last when a service is searched for.

In CAPL the ECU qualifier for an additional description's service is written as
`<ECU qualifier of „Master“ description>.<ECU qualifier of Additional Description>`
(e.g. `Door.BasicDiagnosticsEcu`).

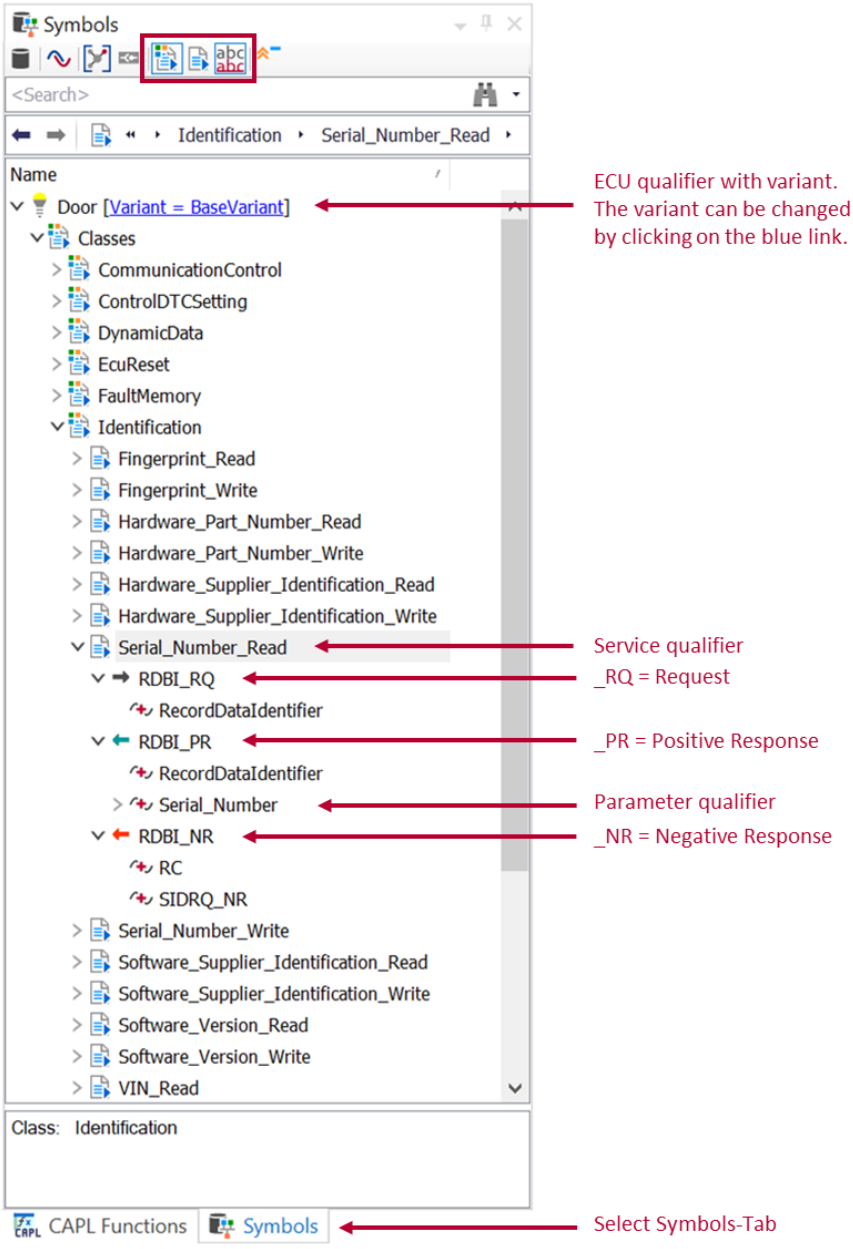
Please find more information about qualifiers in the chapters 4 and _Addressing_the_ECU5.

4 About qualifiers and short names

Each diagnostic object is identified by a qualifier.

The qualifier is in contrast to the name language independent (always indicated in English). In CDD files this qualifier is called a shortcut qualifier. In PDX files the qualifier is equivalent to the ODX short name. The ODX long name cannot be used here.

For programming with CAPL, these qualifiers and objects can easily be accessed in the CAPL browser using the **Symbols** explorer which can be switched to Diagnostics



Please use the symbols in the toolbar to change the display of the diagnostic elements in the tree view. Depending on the selection either the **Diagnostic Classes** or **Diagnostic Services** will be displayed. The diagnostic classes and services can be displayed either with the name or the qualifier by selecting abc abc.

There are 3 types of diagnostic objects which can be referenced in CAPL:

- ECU
- Services
- Parameters

Each of these objects has its own qualifiers which must be used in the various diagnostics functions or in the definition of diagnostics CAPL objects. There is an ECU qualifier that is used to address a specific ECU. For each ECU, there are several diagnostic classes available, but these classes are only used for obtaining a better structure, in CAPL they are not required.

Each class consists of related services (e.g. class Fault Memory contains Fault Memory Services), each service has its own service qualifier. Please note that the read and write services for a specific data identifier have different qualifiers, the basic name of the service appended with a **_Read** or **_Write** supplement. Both parts together constitute the qualifier, for example Serial_Number_Read or Serial_Number_Write.

Again, for a better structure, the qualifiers for a service are divided into 3 categories, consisting of the Mnemonic for the underlying service (e. g. RDBI for **ReadDataByIdentifier**) and an appendix for Positive Response (**_PR**), Negative Response (**_NR**) and Request (**_RQ**).

Underneath these structuring elements, you will find the parameter qualifiers for the respective service. For accessing these parameters, these qualifiers are required.

To use these qualifiers, just drag and drop them from the Symbol Explorer into the CAPL browser's editing window.

5 Addressing the ECU

As you can access multiple ECUs with CANoe\CANalyzer, you need to address the ECU that is targeted by the diagnostic CAPL function calls.

This is done by using the ECU qualifier in `on diag..` event handlers and `DiagRequest/DiagResponse` objects.

The ECU qualifier is specified in the diagnostics configuration (see chapter 3).

DiagRequest object example

```
diagRequest Door.Serial_Number_Write req;
// When sent, the request will address the Door ECU
```

Event handler example

```
on diagResponse Door.Serial_Number_Write
{
  // Triggers when response from Door ECU is received
}
```

6 Creating and sending a request

First you have to define a diagnostic request object which can be done by dragging the service qualifier from the Symbol Explorer to the CAPL browser's editing window.

The data type **DiagRequest** can be seen as kind of a „struct“ object. It represents and contains all the parameters of the request. The **DiagSendRequest** function sends the request using the **DiagRequest** object which is passed as function parameter

```
diagRequest Door.Serial_Number_Read myRequest;
// Create object for the "Serial Number Read" service

diagSendRequest(myRequest);
// Send Request
```

If you want to send a request which is not specified in the diagnostics description, you can create and send the complete request (including SID and subfunction) on raw level.

After the declaration of the **diagRequest** object it is necessary to adjust the size of the service using the the **DiagResize** function. Please find more information about working on raw level in chapter 15.

Find below a sample for this use case.

```
byte request[3] = {0x22, 0xF1, 0x8C};
diagRequest Door.* req;

diagResize(req, elCount(request));
diagSetPrimitiveData(req, request, elCount(request));

diagSendRequest(req);
```

7 Setting the parameters of a request

In many cases a service contains one or more parameters. Before sending a request, you must set its parameters. Parameters can be set using the following methods:

- numeric (physical values: direct values or with applying a conversion rule or formula)
- raw (raw values: as transmitted on the bus byte by byte)
- symbolic (for numeric values which are represented by text tables)

Depending on the type and size of a parameter, you have to use a specific **SetParameter** function:

- for data types with a size of more than 4 bytes (e.g. VIN) or if you want to set the raw value: use **DiagSetParameterRaw**
- for data types, up to 4 bytes use **DiagSetParameter**. The distinction if a parameter is treated as a symbolic or numeric one is done by observing the parameter type. A char[] parameter type is the indicator for a symbolic value, a double type is used for numerical parameters. It is also possible to use **DiagSetParameterRaw** for raw access .
- for avoiding the use of a non-available numerical value or misspelled symbolic parameter values, you should check the return code of the **DiagSetParameter** functions to see if the parameter has been set successfully before sending a request (return value 0 means success and a negative return value means error).

```
diagRequest Door.Serial_Number_Write req;
byte serialNumber[13] = {
0xFF,0xFF,0xFF,0xFF,0xFF,
0xFF,0xFF,0xFF,0xFF,0xFF,
0xFF,0xFF,0xFF
};
long ret;

write("------ Setting of a raw parameter ------");
ret = diagSetParameterRaw(
req, "SerialNumber", serialNumber,
elCount(serialNumber)
);

if(ret>=0)
{
    ret = diagSendRequest(req);
    if(ret>=0)
      write("Request has been (partially) sent");
    else
      write("Could not sent request");
}
else
{
    write("Could not set parameter");
}
```

You can set all the parameters of a request subsequently before finally using the **DiagSendRequest** function to send the request.

If the service contains a parameter with an iterative data type (like a list of DTCs), in order to set the parameter within the iteration it is necessary to use **diagSetComplexParameter** or **diagSetComplexParameterRaw**.

If the service contains a parameter with a variable length, it might be necessary to adjust the size of the service using the **DiagResize** function.

8 Receiving the response and reading the response parameters

After sending a request, you can receive and evaluate the response. For this purpose CANoe/CANalyzer provides the **on diagResponse** event handler. For CANoe test modules there exists a different approach. Please refer to chapter 12.

This event handler is only called when the request has been sent from CAPL.
Requests sent by the Diagnostic Console will not trigger this event handler.

You can read the parameters of the received response using the **DiagGetParameter** and **DiagGetParameterRaw** functions. Like their **DiagSet** counterparts from chapter 7, the use depends on the size and type of the parameters.

You can check whether the received response is a positive or a negative response by using **DiagIsPositiveResponse** or **DiagIsNegativeResponse**.

Parameters can be read by using the following methods:

- numeric (physical values: direct values or with applying a conversion rule or formula)
- raw (raw values: as transmitted on the bus byte by byte)
- symbolic (for numeric values which are represented by text tables)

Depending on the type and size of a parameter, you must use a specific **GetParameter** function:

- for data types with a size of more than 4 bytes (e.g. VIN) or if you want to read the raw value: use **DiagGetParameterRaw**.
- for data types up to 4 bytes: use **DiagGetParameter**. The distinction if a parameter is treated as a symbolic or numeric one is done by observing the parameter type: a char [] parameter type is the indicator for a symbolic value, a double type is used for numerical parameters.
  It is also possible to use **DiagGetParameterRaw** for raw access**.**

There are different types of responses possible:

- no additional parameters (except service ID and data identifier or subfunction)
- additional parameters which can be read as described above
- a negative response code (NRC) in case a negative response has been received.
  The NRC can be read using **diagGetResponseCode** or **diagGetLastResponseCode**.

```
on diagResponse Door.Serial_Number_Read
{
  long ret;
  byte serialNumber[13];

  write("------ Reading of a raw parameter ------");
  if(diagIsPositiveResponse(this))
  {
    ret = diagGetParameterRaw(this,
          "SerialNumber",
          serialNumber, elCount(serialNumber));

    if(ret>=0)
      write("Serial number is (hex): %02X %02X %02X etc.",
    serialNumber[0], serialNumber[1], serialNumber[2]);

    else
      write("Could not retrieve parameter");
  }
  else
  {
    write("Negative response code: 0x%02X",
    diagGetResponseCode(this));
  }
}
```

If the response contains a parameter with an iterative data type (like a list of DTCs), in order to read the parameter`s value within the iteration it is necessary to use **diagGetComplexParameter** or **diagGetComplexParameterRaw**. See chapter 9 for more information.

9 Reading the fault memory

The fault memory returns a list of DTCs and the associated status bits. Reading this list (which normally contains iterations) requires the use of the **DiagGetComplexParameter** (for data types up to 4 bytes) or **DiagGetComplexParameterRaw** (for data types with a size of more than 4 bytes) function. It is essential to pick the correct qualifiers of both the list itself and the subelements, i.e. the DTC and the status byte. By reading the DTC symbolically you can obtain the DTC text.

To read all DTCs, you have to get the number of parameter (DTC) iterations by using the function **diagGetIterationCount** (available since CANoe 9.0 SP3) .

For reading DTC by status mask:

```
on key '7'
{
  diagRequest Door.FaultMemory_ReadAllIdentified req;
  diagSetParameter(req, "DtcStatusMask", 0x09); // Set the status mask
  diagSendRequest(req);
}

on diagResponse Door.FaultMemory_ReadAllIdentified
{
  long length;
  byte StatusByte;
  byte bit;
  char text[200];
  int i;
  dword DTC;

  if(0 != diagIsPositiveResponse(this))
  {
    length = diagGetIterationCount(this, "ListOfDTC"); // Get then number of iterations
    if(length >= 0 )
    {
      write(" ");
      write("Read All identified DTCs:");
      write("-------------------------");

      StatusByte = diagGetParameter(this, "DtcAvailabilityMask");
      write("Status Availability Mask: 0x%02X", StatusByte);

      // Get the symbolic and numeric value of one status availability mask bit
      bit = diagGetParameter(this, "DtcAvailabilityMask.TestFailed");
      diagGetParameter(this, "DtcAvailabilityMask.TestFailed", text, elCount(text));

      write("Test failed bit: %s - 0x%02X", text, bit);
      write("-------------------------");

      for(i=0;i<length;i++) // iterate through all DTCs
      {
        // Get the symbolic and numeric value of the DTC
        DTC = diagGetComplexParameter(this,"ListOfDTC", i, "DTC");
        diagGetComplexParameter(this, "ListOfDTC", i, "DTC", text, elCount(text));
        // Get the DTC status byte as numerical value
        StatusByte = diagGetComplexParameter(this,"ListOfDTC", i, "StatusOfDtc");

        write("DTC 0x%06X - %s",DTC,text);
        write("StatusByte: 0x%02X",StatusByte);

        // Get the symbolic and numeric value of one status byte bit
        bit =
        diagGetComplexParameter(this,"ListOfDTC", i,"StatusOfDtc.TestFailed");

        diagGetComplexParameter
        (this, "ListOfDTC", i, "StatusOfDtc.TestFailed", text, elCount(text));

        write("Test failed: %s - 0x%02X",text,bit);
        write("-------------------------");
      }

     }
     else
```

```
    {
      write("Error retrieving iteration length: %d", length);
    }

  }
  else
  {
   diagGetParameter(this, "RC", text,
   elCount(text)); // get the symbolic value of the response code

   write("Negative response received.\nNegative response code: 0x%02X - %s",
  (byte)DiagGetResponseCode(this), text);
  }
}
```

10 Reading extended data records and snapshot data of the fault memory

Extended data records and snapshot data records can be different for each DTC, and there can be one or more different record types for both data structures. The assignment of the record numbers or types is done using multiplexor components which determine the data structure used.

Generally, the extended data record number or the snapshot number is the multiplexor which determines the contents of the associated extended data record or the snapshot record. For reading such data, you must observe the record number and then read the corresponding parameters for each record type. If you want to write such data structures (for ECU simulation), you have to set the multiplexor parameter first (which will adapt the data structure immediately) and then the parameters of the records. Extended data records or snapshot records can also be implemented as lists (iterations) of such records, similar to DTC lists. In this case the complex parameter CAPL functions must be used (see chapter 9).

The example below for these data structures is not universal, there are different ways to model and implement such data structures in diagnostic description files. Especially for extended data records, the example is not typical as it does only provide access to one specific extended data record, not to all at once.

Note: The parameter qualifiers in the following examples depend on the used diagnostic description, therefore you most likely need to adapt them to your requirements.

For reading extended data records:

```
on key '8'
{
  diagRequest Door.FaultMemory_Read_extended_data req;
  dword DTC = 0x402011;
  byte extendedDataRecNumber = 0x01;

  write("Reading extended data records:");
  write("------------------------------");
  diagSetParameter(req, "DTC", DTC); // Set the specific DTC

  // Read extended data record "Occurrence Counter"
  diagSetParameter(req,
  "Record_Numbers",
  extendedDataRecNumber);

  diagSendRequest(req);
}
```

```
on diagResponse Door.FaultMemory_Read_extended_data
{
  dword DTC;
  byte statusByte;
  long iCount;
  int i;
  byte recNo;
  int ocCounter;
  int agCounterDTC;
  char buffer[100];

  if(diagIsPositiveResponse(this))
  {
    DTC = diagGetParameter(this, "DTC"); // Get the DTC
    statusByte = diagGetParameter(this, "Status_Of_Dtc"); // Get the status byte

    write("DTC: %X", DTC);
    write("StatusByte: %X", statusByte);

    // Get the number of extended data record iterations
    iCount = diagGetIterationCount(this, "ListOfDTCExtendedDataRecord");

    for(i=0;i<iCount;i++) // iterate through all data records
    {
      // Get the data record number (multiplexor) as numeric and symbolic value
      recNo = diagGetComplexParameter(this, "ListOfDTCExtendedDataRecord", i,
              "Record_Numbers");
      diagGetComplexParameter(this, "ListOfDTCExtendedDataRecord", i,
              "Record_Numbers", buffer, elCount(buffer));
      write("Extended Data Record: 0x%02X (%s)", recNo, buffer);

      if(recNo == 0x01) // Extended data record 0x0A
      {
        ocCounter = diagGetComplexParameter(this, "ListOfDTCExtendedDataRecord",
                    i, "Occurrence_Counter");
        write("OccurenceCounter: %d", ocCounter);
      }

      if(recNo == 0x02) // Extended data record 0x0B
      {
        agCounterDTC = diagGetComplexParameter(this, "ListOfDTCExtendedDataRecord",
                       i, "Aging_Counter");
        write("DTC Aging Counter: %d", agCounterDTC);
      }
    }
  }
  else
  {
    // get the symbolic value of the response code
    diagGetParameter(this, "RC", buffer, elCount(buffer));
    write("Negative response received.\nNegative response code: 0x%02X - %s",
          (byte)DiagGetResponseCode(this), buffer);
  }

}
```

For reading snapshot data records:

```
on key 's'
{
  diagRequest Door.FaultMemory_ReadEnvironmentData req;
  dword DTC = 0x402011;
  byte snapshotNumber = 0x10;

  write("Reading snapshot records:");
  write("------------------------------");

  diagSetParameter(req, "DTC", DTC); // Set the DTC

  // Read snapshot record "First Occurrence"
  diagSetParameter(req,
  "Record_Numbers",
  snapshotNumber); // Set the snapshot record number

  diagSendRequest(req);
}
on diagResponse Door.FaultMemory_ReadEnvironmentData
{
  dword DTC;
  byte statusByte;
  byte recNo;
  byte NoIDs;
  byte snapshotRecord[21];
  float wheelSpeedFR;
  float valveVoltageFR;
  int iCount, index;
  char buffer[100];

  if(diagIsPositiveResponse(this))
  {
    DTC=DiagGetParameter(this,"DTC"); // get DTC
    statusByte=DiagGetParameter(this, "StatusOfDTC");  // get status byte

    write("DTC %06X",DTC);
    write("StatusByte: %02X",statusByte);

    iCount = diagGetIterationCount(this,
             "ListOfDTCSnapshotRecord"); // Get the number of iterations

    for(index=0;index<iCount;index++) // iterate through all data records
    {
      // Get the data record number (multiplexor) as numeric and symbolic value
      recNo=DiagGetComplexParameter(this,"ListOfDTCSnapshotRecord",index,
                                    "Record_Numbers");
      DiagGetComplexParameter(this,"ListOfDTCSnapshotRecord",index,
                                    " Record_Numbers ",
                                     buffer, elCount(buffer));
      write("Snapshot Data Record: 0x%02X (%s)", recNo, buffer);

      if(recNo == 0x10) // Snapshot data record 0x10
      {
        // get the number of identifiers
        NoIDs=DiagGetComplexParameter(this,"ListOfDTCSnapshotRecord",index,
                                      "DtcSnapshotRecordNumberOfIdentifiers");
        // get the data of this snapshot record
        wheelSpeedFR=DiagGetComplexParameter(this,"ListOfDTCSnapshotRecord",index,
                                "Wheel_Speed_FR");
        valveVoltageFR=DiagGetComplexParameter(this,"ListOfDTCSnapshotRecord",index,
                                      " Valve_Voltage_FR");
```

```
      write("NumberOfIDs: %d",NoIDs);
      write("DID 0x%04X - Wheel Speed FR: %.1f",
      (word)DiagGetComplexParameter(this, "ListOfDTCSnapshotRecord",
      index,"Wheel_Speed_ID"), // display the DID number in the write window
      wheelSpeedFR);

      write("DID 0x%04X - Valve Voltage FR: %.1f",
      (word)DiagGetComplexParameter(this,"ListOfDTCSnapshotRecord",
      index,"Brake_Pedal_ID"), // display the DID number in the write window
      valveVoltageFR);
      write("---------------------------------------");
    }
  }
 }
 else
 {
   diagGetParameter(this, "RC",
   buffer, elCount(buffer)); // get the symbolic value of the response code
   write("Negative response received.\nNegative response code: 0x%02X - %s",
         (byte)DiagGetResponseCode(this), buffer);
 }
 }
```

11 Security access with Seed & Key DLL

The seed & key procedure can be handled using the **DiagGetParameter/DiagSetParameter** functions and the **DiagGenerateKeyFromSeed** or **DiagStartGenerateKeyFromSeed** function which will execute the secret key calculation algorithm implemented in the seed & key DLL. For seeds and keys with more than 4 bytes it is necessary to use the **DiagGetParameterRaw/DiagSetParameterRaw** functions as shown in the example below.

DiagGenerateKeyFromSeed or **DiagStartGenerateKeyFromSeed** requires the security level (that is the subfunction of the 0x27 UDS service, for example 0x27 03/04 is used for level 3).
If the **ipOption** parameter is not used, it should be an empty string.

Which function to use depends on the length of the key computation. If the computation is guaranteed to take significantly less than 1 ms, **DiagGenerateKeyFromSeed** may be used. Otherwise **DiagStartGenerateKeyFromSeed** in combination with the callback **_Diag_GenerateKeyResult** should be used.

For CANoe test modules there are different functions (e.g. **TestWaitForGenerateKeyFromSeed** or **TestWaitForUnlockEcu**) available. For a general introduction of diagnostics in test modules please look into chapter 12. Templates for the seed & key DLL (Visual Studio, C++) can be found in this directory:

```
C:\Users\Public\Documents\Vector\CANoe\Sample Configurations [CANoe-
Version]\CAN\Diagnostics\UDSSystem\SecurityAccess
```

The templates are supplied with CANoe.

Please add the seed & key DLL to the CANoe/CANalyzer Diagnostic Configuration as explained in chapter 3

For Security Level 1:

```
on key 'k'
{
  diagRequest Door.SeedLevel1_Request req;
  long ret;

  write("------ Security access level 1 ------");

  ret=DiagSendRequest(req); // send seed request
  if(ret>=0)
      write("Request seed level 1 has been (partially) sent");
  else
```

```
      write("Error while sending request. Error code: %ld", ret);

}

on diagResponse Door.SeedLevel1_Request
{
  byte seedArray[4];
  long ret;
  char buffer[100];

  if(diagIsPositiveResponse(this))
  {
    ret=DiagGetParameterRaw(this,"SecuritySeed",seedArray, elcount(seedArray));
    if(ret>=0)
    {
      write("Seed is (hex) %X %X %X %X",
            seedArray[0],seedArray[1],seedArray[2],seedArray[3]);

      // calculate the key using the received seed: security level 1, variant
"Common"
      ret=DiagStartGenerateKeyFromSeed("Door",seedArray,
          elcount(seedArray),1,"Common","");

      if(ret==0) // Key computation was started
      {
        write("Key computation for security level 1 was started");
      }
      else // errors occured while calculating the key
      {
        write("Error code %ld during key calculation",ret);
        if(ret==-84)
          write("invalid security level");
        if(ret==-86)
            write("buffer too small");
      }
    }
    else
    {
      write("Could not retrieve parameter");
    }
  }
  else
  {
    diagGetParameter(this, "RC", buffer,
     elCount(buffer)); // get the symbolic value of the response code
    write("Negative response received.\nNegative response code: 0x%02X - %s",
    (byte)DiagGetResponseCode(this), buffer);
  }
}

// Callback for "diagStartGenerateKeyFromSeed"
_Diag_GenerateKeyResult( long result, BYTE computedKey[])
{
  diagRequest Door.KeyLevel1_Send reqKey;
  int i;
  long ret;
  char keyBuffer[4], stringBuffer[100];

  if(result==0) // if key has been calculated
  {
    // To output the key in one line, a composite string has to be build
    snprintf(stringBuffer, elCount(stringBuffer),""); // Clear the string buffer
    strncat(stringBuffer, "Key is (hex): ",
    elCount(stringBuffer)); // Begin to build composite string
    for(i=0;i<elCount(computedKey);i++)
    {
      snprintf(keyBuffer, elCount(keyBuffer), "%02X ",
      computedKey[i]); // Print current key byte into key buffer
      strncat(stringBuffer, keyBuffer,
       elCount(stringBuffer)); // Add current key byte to the string buffer
    }
    write(stringBuffer); // Output the string buffer (the key) in one line

    ret=DiagSetParameterRaw(reqKey,"SecurityKey",
    computedKey,elcount(computedKey)); // put the key into the send key service
    if(ret>=0)
```

```
      DiagSendRequest(reqKey); // send the key
    else
      write("Could not set key-parameter");
  }
  else // errors occured while calculating the key
  {
    write("Error code %ld during key calculation",result);
    if(result==-84)
      write("invalid security level");
    if(result==-86)
        write("buffer too small");
  }
}
```

Simple Seed & Key DLL source code:

```
KEYGENALGO_API VKeyGenResultEx GenerateKeyEx(
  const unsigned char* iSeedArray,     /* Array for the seed [in] */
  unsigned int         iSeedArraySize, /* Length of the array for the seed
                                          [in] */
  const unsigned int   iSecurityLevel, /* Security level [in] */
  const char*          iVariant,       /* Name of the active variant [in]*/
  unsigned char*       ioKeyArray,     /* Array for the key [in, out] */
  unsigned int         iKeyArraySize,  /* Maximum length of the array
                                          for the key [in] */
  unsigned int&        oSize           /* Length of the key [out] */
)
{
  // check the input parameters
 if (iKeyArraySize<2)    // check if return buffer is sufficiently large
   return KGRE_BufferToSmall;

  if (iSeedArraySize!=2) // check if seed is 2 bytes, this algorithm works
                         // with 2 bytes seed and 2 bytes key only
    return KGRE_UnspecifiedError;

  switch(iSecurityLevel)
  {
    case 1: // algorithm for security level 1: swap the 2 seed bytes
      ioKeyArray[0]=iSeedArray[1];
      ioKeyArray[1]=iSeedArray[0];
      oSize=2;
      break;

    case 3:  // algorithm for security level 3: invert the seed bytes
      for(unsigned int i=0;i<iSeedArraySize;i++)
        ioKeyArray[i]=~iSeedArray[i];
        oSize=2;
        break;

    default: // any other level is not handled by this DLL
      return KGRE_SecurityLevelInvalid;
      break;
  }
  return KGRE_Ok;
}
```

12 Diagnostics in test modules

Creating diagnostic tests with simulation nodes can lead to CAPL code of which the control flow is jumping from one event handler to the other and is not quite easy to understand and follow. For this reason, CANoe's Test Feature Set (TFS) offers the possibility of creating test sequences by providing **TestWaitFor** functions which are not available in simulation nodes because of the real time architecture of CANoe.

For diagnostic tests with its request-response pairing, these test sequences are very suitable elements. After sending a request, you should make sure the request has been sent by using the function **TestWaitForDiagRequestSent**. Afterwards you wait for the response using the **TestWaitForDiagResponse** function, the received parameters of the response can be obtained with the **diagGetRespParameter** function.

The TFS offers test report features that provide HTML reports of the tests. Diagnostic objects can be written into the test report using the **TestReportWriteDiagObject** or **TestReportWriteDiagResponse** function.
It is important to evaluate the response of the **TestWaitForDiagResponse** function, as it could hint to either the timeout specified in this function or the diagnostic P2 or P2 extended timeout. See KnowledgeBase article: Handling of TestWaitFor DiagResponse Function.

```
void MainTest ()
{
  TestCase1_ReadSpeedFR();
}

testcase TestCase1_ReadSpeedFR()
{
  DiagRequest Door.Wheel_Speed_Read req;
  long ret,ret2,ret3;
  char buffer[100];
  float wheelSpeedFR;

  // Write the current teststep to the testreport
  teststep(0, "1.0", "Read RPM of wheel FR");

  // Get the name of the diagnostic object
  diagGetObjectName(req, buffer, elCount(buffer));
  diagSendRequest(req);

  // Wait until the request has been completely sent
  ret = TestWaitForDiagRequestSent(req, 5000);

  if(ret==1) // Request sent
  {
    TestReportWriteDiagObject(req); // Write the request-Object to the testreport
    write("Request has been successfully sent");
    // Wait for a response, here for 5000ms. Note: This is no P2 timeout!
    ret2=TestWaitForDiagResponse(req,5000);

      if(ret2==1) // Response received
      {

        ret3=DiagGetLastResponseCode(req); // Get the code of the response
        if(ret3==-1) // Is it a positive response?
        {
          // get the response`s parameter in physical format
          wheelSpeedFR=DiagGetRespParameter(req, 1, "Wheel_Speed_FR");
          write("Speed of wheel FR is %.0f RPM",wheelSpeedFR);
          TestReportWriteDiagResponse(req); //Write the response object to the testreport
          // teststep accomplished
          testStepPass(0, "1.0", "Speed of wheel FR is %.0f RPM", wheelSpeedFR);
        }
        else // It is a negative response
        {
          write("Negative Response, NRC: 0x%02X",(byte)ret3);
          TestReportWriteDiagResponse(req); //Write the response object to the testreport
          // teststep failed
          testStepFail(0, "1.0", "Negative Response, NRC: 0x%02X",(byte)ret3);
        }
        if(ret2==0) // timeout. no response received
        {
          write("Timeout specified in TestWaitForDiagResponse expired");
          testStepFail(0, "1.0", "Timeout specified in TestWaitForDiagResponse expired");
        }
        if(ret2<0) // error e.g. transport protocol level
        {
          if(ret2==-92) // This is the error code for P2 or P2* timeouts
          {
            write("TP level error %d, probably P2 or P2* timeout", ret2);
            testStepFail(0, "1.0", "TP level error %d, probably P2 or P2* timeout", ret2);
          }
          else
```

```
        {
          write("Error %d in the diagnostic or transport layer", ret2);
          testStepFail(0, "1.0", "Error %d in the diagnostic or transport layer", ret2);
        }
      }
  }
  else
  {
    if(ret==0)
    {
      write("Timeout expired while trying to send request %s", buffer);
      testStepFail(0, "1.0", "Timeout expired while trying to send request %s", buffer);
    }


    if(ret<0)
    {
      write("Internal error %d occured while trying to send request %s", ret, buffer);
      testStepFail(0, "1.0",
       "Internal error %d occured while trying to send request %s", ret, buffer);
    }
  }
}
```

13 Simulating an ECU

For common usecases it is simple to setup an ECU simulation (See the step-by-step instructions below). Only in case of more complex ECU simulations (e.g. fault injection on TP level) the CAPL Callback Interface (CCI) as well as the `OSEK_TP.DLL` must be added to the simulation node.

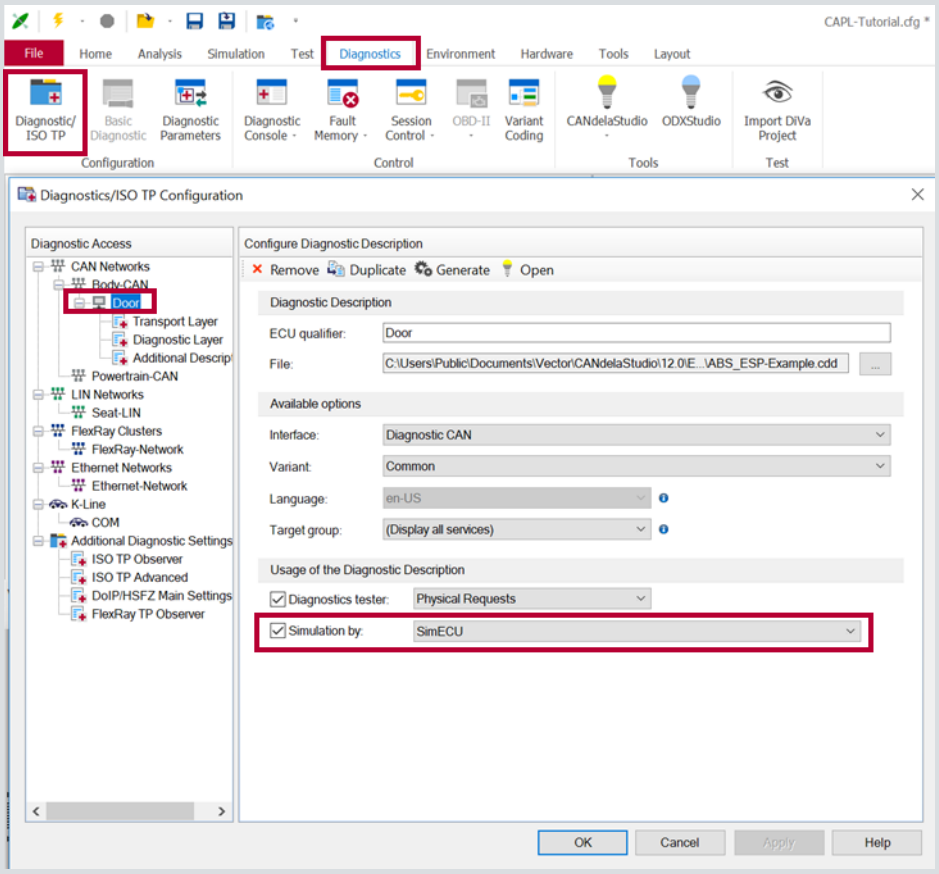
Find more information about the CCI in the Application Note **AN-IND-1-012_CAPL_Callback_Interface**, which is provided in the Doc-folder of your CANoe installation

Step-by-step instructions: How to setup an ECU simulation

> Open CANoe and add a CAPL node in the Simulation Setup.

> Add the diagnostic description for the ECU to be simulated under **Configuration | Diagnostics/ISO-TP configuration**.

> Activate the check box **Simulation by** and select the previously created CAPL node.



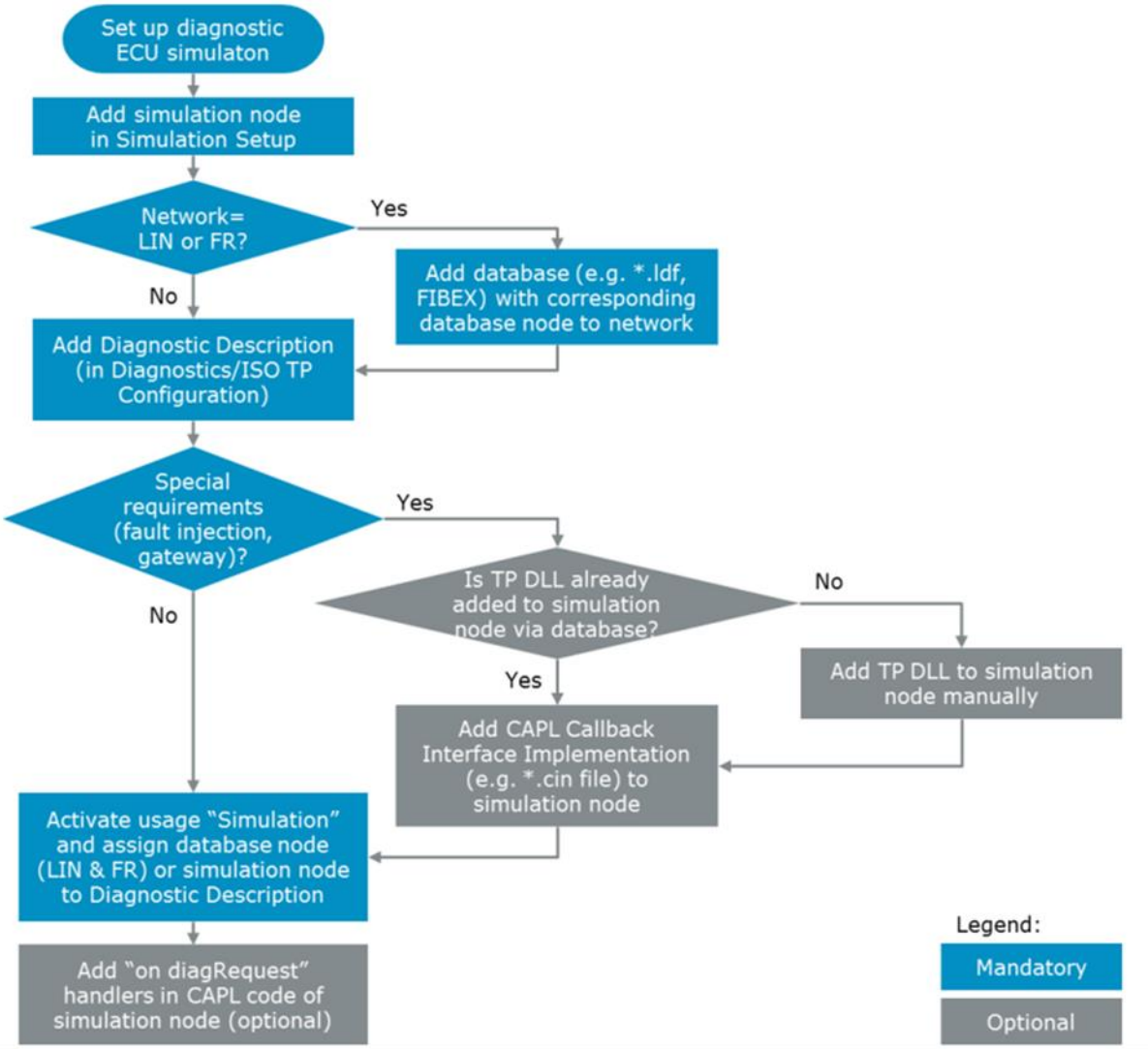
> Right-click on the node in the Simulation Setup and select **Configuration**.

> Assign a CAPL-File (`*.can`) to this node under **Node specification**

> Edit the CAPL code:

  In the **on diagRequest** section you have to create the event handlers for the diagnostic requests sent to this simulated ECU. You can use all the CAPL **DiagGetParameter** and **DiagSetParameter** functions to read the request's parameters and to set the parameters for the response. Use **DiagSendResponse** to send your response.

Please find below a chart which shows the steps outlined above in a more compact way.
Note: For LIN and FlexRay ECUs, adding a database (`*.ldf, *.fibex`) is mandatory.



Workflow when setting up a diagnostics ECU simulation

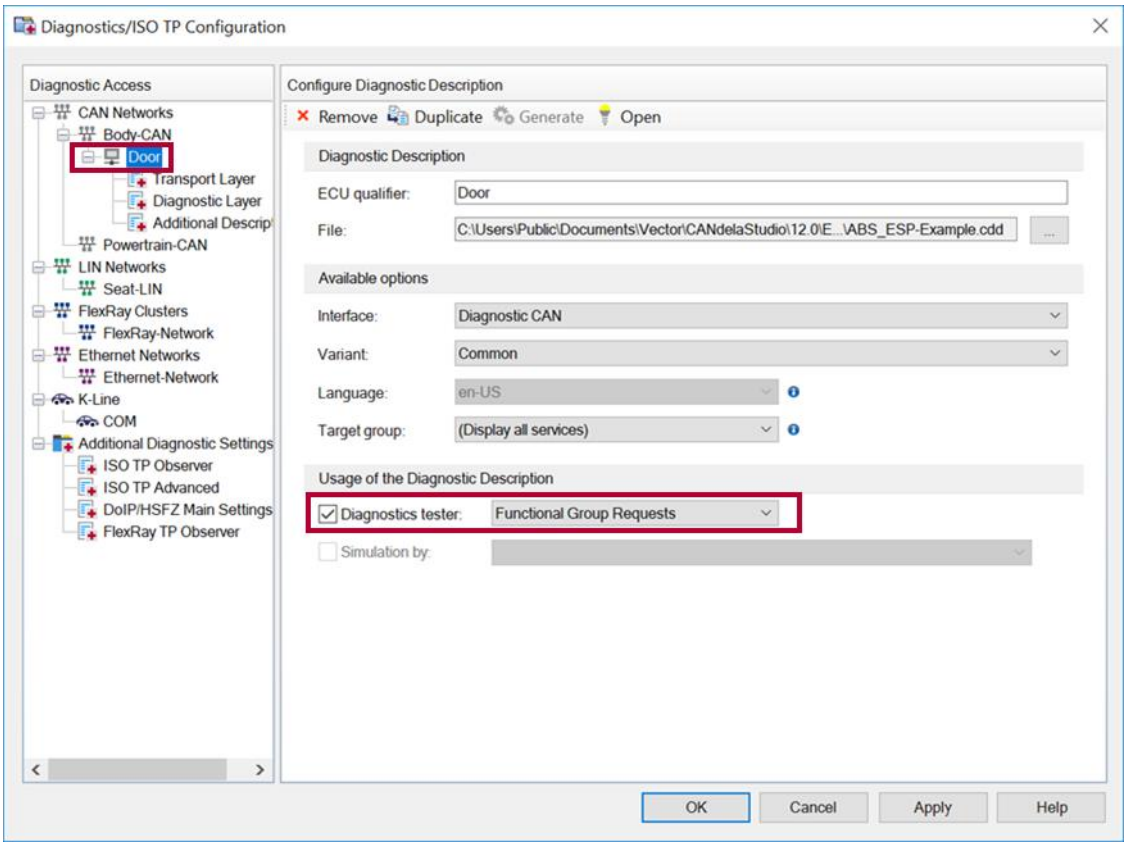
To give you an idea how the CAPL code of an ECU simulation might look like, please note the following code example. The simulated ECU does only respond to service **Serial Number Read** with some data and sends a simple positive response otherwise.

```
on diagRequest *
{
  diagResponse this resp;
  diagSendPositiveResponse(resp);
}

on diagRequest Door.Serial_Number_Read
{
  diagResponse Door.Serial_Number_Read resp;
  byte serialNumber[13] = {
  0xFF,0xFF,0xFF,0xFF,0xFF,
  0xFF,0xFF,0xFF,0xFF,0xFF,
  0xFF,0xFF,0xFF
  };
  long ret;

  ret = diagSetParameterRaw(resp,
       "Serial_Number", serialNumber, elCount(serialNumber));

  if(ret >= 0)
    diagSendResponse(resp);
}
```

14 Sending functional requests

If there is functional addressing defined in the diagnostic description, you can send a request to more than one ECU.

Please open the diagnostic configuration and select under **Usage of the diagnostics description** the entry **Functional Group Requests**.



From now on, each diagnostic request will be sent using the functional Request ID. To be able to receive and analyze the response please add another diagnostic description which contains the same functional Request ID. Usually it is sufficient to make a copy of the diagnostic description file and add the copy. The Diagnostics tester of the second description must be set to **Physical Requests**.

It is also possible to send a request with functional addressing although the usage of the diagnostic description is set to Physical Requests. For this the CAPL function `diagSendFunctional` can be used.
When only one functional request should be sent, using `diagSendFunctional` might be simpler than to reconfigure the usage of the diagnostic description.

But keep in mind that only the response from the diagnostic descriptions target will be detected, since the usage is set to Physical Requests. The response from another ECU cannot be received.

When waiting for the response, keep in mind that there will probably be more than one response, so you have to keep waiting for a response after the reception of a response as long as a P2 timeout has not occurred (because after reception of a response, the P2 timer is reset).

15 Manipulating diagnostic data on raw level

If the services defined in the diagnostic description file does not fit your purposes or for creating and checking faults in tests you can access a service on the raw byte level, the so-called primitive level. Here you have a stream of bytes. CANoe offers the functions **DiagSetPrimitiveData** and **DiagSetPrimitiveByte** to set the complete service or just a single byte, and the functions **DiagGetPrimitiveData** and **DiagGetPrimitiveByte** to read the whole primitive or a single byte of the service.

```
diagrequest Door.* req;
diagrequest Door.* req2;
byte rawData[2]={0x10,0x99};

diagResize(req, elCount(rawData));
DiagSetPrimitiveData(req,rawData,elcount(rawData));
DiagSendRequest(req);

// is equivalent to:

diagResize(req2, 2);
DiagSetPrimitiveByte(req2,0,0x10);
DiagSetPrimitiveByte(req2,1,0x99);
DiagSendRequest(req2);
```

You could change a byte (e.g. a subfunction or a data identifier) to a value that does not exist and check how the ECU responds to this request. The size of a request or response can be calculated by the **DiagGetPrimitiveSize** function. As a service might have been changed in a way that CANoe does not recognize the response (as it is not defined in this way in the diagnostic description), you must use the **on diagResponse *** event handler to receive the response (or the request using **on diagRequest *** in an ECU simulation). This handler will only be triggered if no other **on diagResponse** event handler fits to the received response.

```
on diagResponse Door.*
{
  byte rawData[4095];

  if((DiagGetPrimitiveByte(this,0)==0x50)
      &&(DiagGetPrimitiveByte(this,1)==0x99))
  {
    write("This is the response to not-defined service 0x10 0x99");
    DiagGetPrimitiveData(this,rawData,elcount(rawData));
    write("%d Bytes received",DiagGetPrimitiveSize(this));
  }
}
```

In general, we advise to use **Additional Diagnostic Descriptions** in combination with **Basic Diagnostics** (see chapter 3) for services not defined in the description, so the main area of application of these functions will be fault injection.

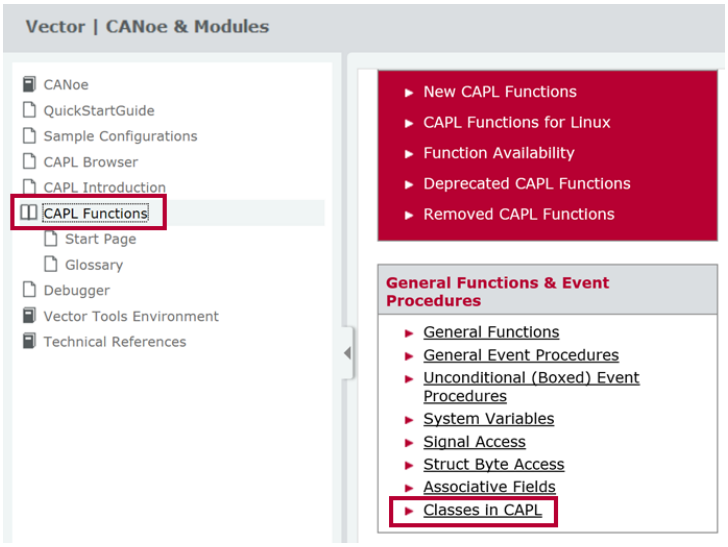
However, for complex services the **Diag…Primitive** functions might be better suitable than **Basic Diagnostics**.

16 Object-oriented programming

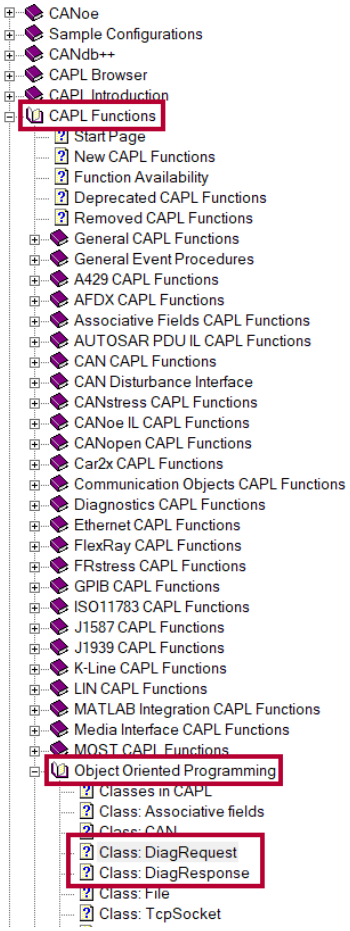
In CAPL the datatypes **diagRequest** and **diagResponse** can also be used like classes in object-oriented programming languages. This means that all diagnostics functions (methods) belonging to the **diagRequest**- or **diagResponse**-class can be called like methods on objects.

The available methods of both classes are listed in CANoe\CANalyzer`s Help.

HTML5 Help (available since CANoe\CANalyzer 14.0):



CHM Help:

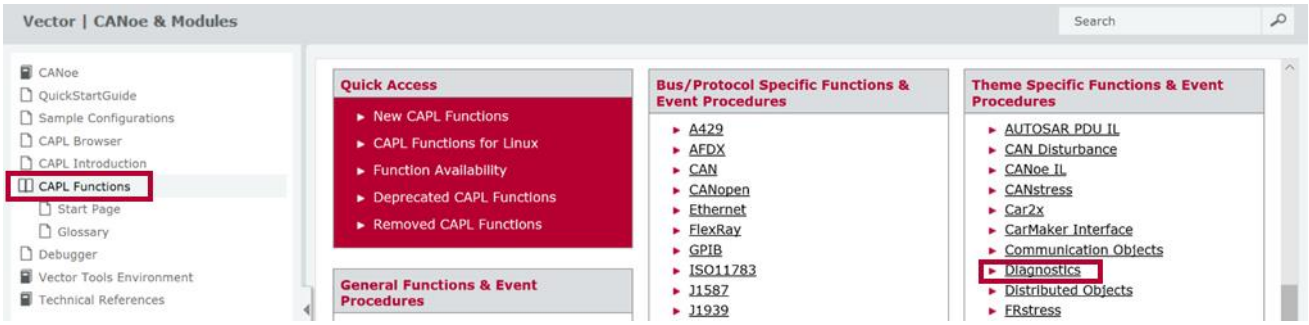


Please find the CAPL example from chapter 7 adapted to object-oriented programming below.

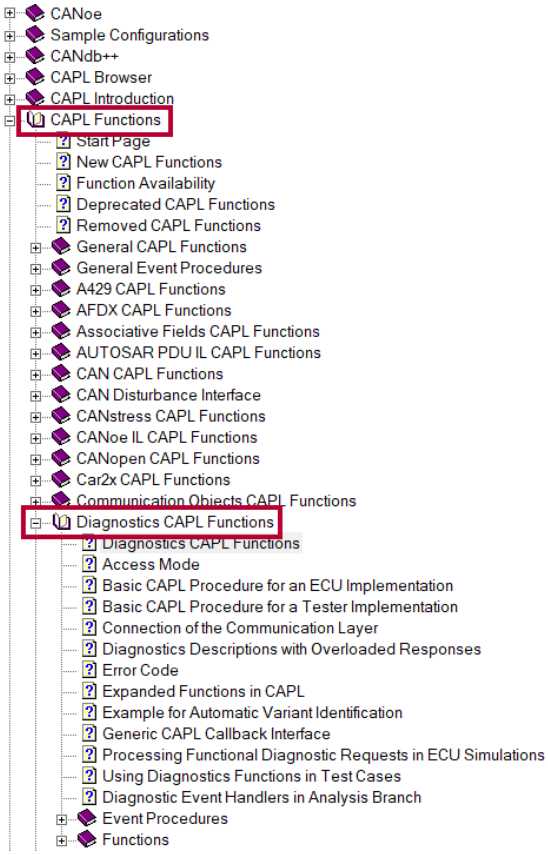
```
diagRequest Door.Serial_Number_Write req;
byte serialNumber[13] = {
0xFF,0xFF,0xFF,0xFF,0xFF,
0xFF,0xFF,0xFF,0xFF,0xFF,
0xFF,0xFF,0xFF
};
long ret;

write("------ Setting of a raw parameter ------");
ret = req.SetParameterRaw("SerialNumber", serialNumber,
elCount(serialNumber)
);

if(ret>=0)
{
  ret = req.SendRequest();
  if(ret>=0)
    write("Request has been (partially) sent");
  else
    write("Could not sent request");
  }
  else
  {
    write("Could not set parameter");
  }
```

17 Where to find more information

Find an explanation of all diagnostic CAPL Functions in the CANoe\CANalyzer`s Help.
HTML5 Help (available since CANoe\CANalyzer 14.0):



CHM Help:



Several diagnostic CAPL samples are available in the KnowledgeBase.

Detailed information about CANoe\CANalyzer`s diagnostic features can be found in the following Application Notes:

- AN-IND-1-001_CANoe_CANalyzer_as_Diagnostic_Tools
- AN-IND-1-004_Diagnostics_via_Gateway_in_CANoe
- AN-IND-1-012_CAPL_Callback_Interface

The Application Notes are provided in the Doc-folder of your CANoe\CANalyzer installation (usually `C:\Program Files\Vector CANoe [version]\Doc`).

18 Contact information

For support related questions please address to the support contact for your country https://www.vector.com/int/en/company/contacts/support-contact/.