POOL

Portable Object Oriented Language

Tutorial: ADX-Library

Revision level: see Revision index

©

SIEMENS VDO Automotive

VDO-Straße 1
12345 Babenhausen
Germany

bdt

Büro für Datentechnik GmbH
D-35418 Buseck
Germany

1 Contents

2 Revision index

Date	**Author**	**Rev.**	**Ref.**	**Type**	**Description**
2003-06-02	Harald Ebert	0.90.00	-	-	Initial Revision

Acronyms:

AIDA

Automotive and **I**ndustrial **D**iagnostic **A**ssistance. System that is used to implement computer supported diagnosis of control modules and bus systems.

BSK

The manufacturer of the AIDA-Systems.

ECU

Electrical **C**ontrol **U**nit.

POOL

Portable **O**bject **O**riented **L**anguage. Object oriented programming language by BSK. POOL is used for programming in the AIDA system.

RDM

BSKD-ECU

SMK

Software **M**ethod **K**it. Description of the programming guidelines by SiemensVDO.

3 Introduction

AIDA is a tool to support diagnosis in the automotive industry. This tutorial introduces the ADX (AIDA Data Exchange) library.

3.1 Where does ADX fit into the AIDA system?

In Fig. 1 you see a POOL application running on a PC communicating with an ECU via CAN bus. As you can see the communication is done via several layers, as there are the application layer, the ADX layer and the AIDA Stack layer (whereby the AIDA Stack layer itself consists of layers too).

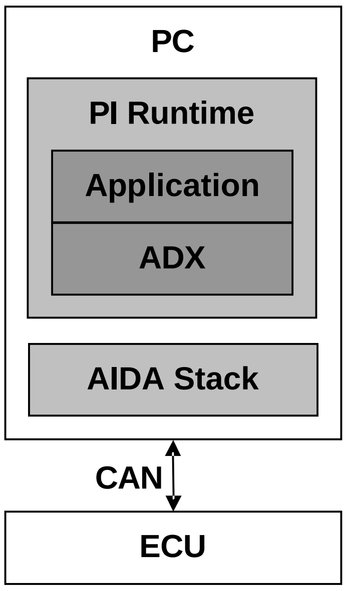

Figure 1: Communication via layers

AIDA uses components with exactly defined interfaces to allow a layered combination of components. This allows an application for instance to communicate with different ECUs using different protocols like KWP2000 or TP 2.0 on different communication media like CAN bus, IEEE 488 or RS232 simply by replacing the corresponding component.

This is nothing new to users who know AIDA, since this feature is provided via the stack components. So what is ADX for?

ADX is a wrapper around the AIDA library allowing using objects to communicate with the ECU instead of calling function. The primary goal is to allow the user to concentrate on the data by treating the data as objects and accessing them via methods. This is a big advantage since you can forget all the communication details that would be necessary to take care of otherwise. Simply put: ADX simplifies development of communication between target and diagnosis program.

Note: The recommended approach is to configure a stack using the Stacker tool, and to store that configuration to the hard drive. This configuration file is then loaded by the application during runtime.

3.2 The Example

AIDA ships with example files. We picked the RDM example to show you how to access an ECU. There reason why we picked this example is that you do not need any special hardware to test it nor do you have to implement a simulator application.

3.2.1 Setting up the example

1. Open the AIDA Studio and create a new project.
2. Import the example files:

• BSKD38K4SIM.aida-cfg	(target stack configuration file)
• bskdsim.pool	(target simulation)
• RDM.aida-cfg	(host stack configuration file)
• tst_bskd.pool	(simulation framework)
• rdm.pool	(actual communication application)
• rdm_sym.pool	(example structures)

3. Set environment variables
   - BSKD38K4SIM_COM_PORT = 2
   - BSKD38K4SIM_COM_ECHOSUPPRESSION = 0
   - RDM_COM_PORT = 1
   - RDM_BDIAG_ECHOSUPPRESSION = 0
4. Connect COM1 with COM2 using a null modem cable
5. Compile the file tst_bskd.pool and run it.
6. Execute the following lines in the Commander
   - oMem.boRd(0x1234,0x20);
   - oMem.vShow;
7. Now you should see the following output:

```
       4  5  6  7  8  9  A  B  C  D  E  F  0  1  2  3
1234: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "................"
1244: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "................"
```

3.2.2 The sample target

The bskdsim.pool example file, which is delivered with the AIDA Studio provides an implementation of a target simulation. The file BSKD38K4SIM.aida-cfg defines the stack for the target simulation. We do not need to understand the code of the target simulation. All we need to know is how to communicate with the target which is explained next.

Memory:

The target has a memory of 64KB and a non volatile memory of 255 byte.

Commands:

`'I'`	returns the identification string.
`'R'`	reads the target's memory (range: 0 – 65535)
`'W'`	writes to the target's memory (range: 0 – 65535)
`'r'`	reads the target's non volatile memory (EEPROM) (range: 0 – 255)
`'w'`	writes to the target's non volatile memory (EEPROM) (range: 0 – 255)
`'G'`	get status
`'S'`	set status
`ncETB`	end diagnosis

Segment size:
8Bytes

3.2.3 A first communication example

Reading and writing the target's memory is one of the most important diagnostic actions. After having set up out test environment successfully let's try to read and write one byte of the target's memory.

A first communication example

```
module TEST;

import AIDA;                           {import the AIDA library}
import NORM;                           {import the NORM library}
import ADX;                            {import the ADX library}

const
  csStackName:  String = 'RDM';       {AIDA stack name}
{global variables}
var
  csStackIdent: CharString;
  bsCmdRd:      ByteString;            {read telegram}
  bsCmdWr:      ByteString;            {write telegram}
  bsResult:     ByteString;            {result telegram}
  oCommu:       ADX_toCommu;           {ADX communication object instance}
  boResult:     Boolean;               {variable to store the return values}

private
{private variables}
var
```

```
21   hStack:       tHandle;            {ECU stack handle}
22   oStack:       AIDA_toStack;       {ECU stack instance}
23   oFormat:      NORM_toFormat;      {ECU data format instance}
24
25 {private procedures}
26 procedure vInitStack(var hStack: tHandle; csFilename: CharString);
27 var
28   dwSLevelMask: DWord;
29 begin
30   hStack := AIDA_hRestoreStack(csFilename, csStackIdent, dwSLevelMask);
31   if hStack = nil then
32   Writeln('Error loading stack');
33   Halt(nExitSysError);
34   endif;
35 end;
36
37 procedure vDeinit;
38 begin
39   oCommu.vDone;
40   oStack.vRemoveEvents;
41   oStack.vDone;
42   if hStack <> nInvHandle then
43     if !AIDA_boDeleteStack(hStack) then
44       Writeln("?RDM – Error deleting stack");
45     endif
46   endif;
47 end;
48
49 begin
50   // initialize object instances
51   oFormat.poInit(NORM_nenBOLittleEndian);
52   vInitStack(hStack, csStackName);
53   oStack.poInit(hStack);
54   oCommu.poInit(@oStack, @oFormat);
55   // initialize read telegram
56   bsCmdRd := Byte(Ord('R'));        {command 'read'}
57   bsCmdRd := bsCmdRd + Byte(0x01);  {length 1}
58   bsCmdRd := bsCmdRd + Byte(0x12);  {address high byte}
59   bsCmdRd := bsCmdRd + Byte(0x34);  {address low byte}
60   // initialize write telegram
61   bsCmdWr := Byte(Ord('W'));        {command 'write'}
62   bsCmdWr := bsCmdWr + Byte(0x01);  {length 1}
63   bsCmdWr := bsCmdWr + Byte(0x12);  {address high byte}
64   bsCmdWr := bsCmdWr + Byte(0x34);  {address low byte}
65   bsCmdWr := bsCmdWr + Byte(0x33);  {data byte}
66 end.
```

First the AIDA library, the NORM library, and the ADX library are included. The use of the different libraries is explained later. Line 8 defines the stack name (RDM.aida-cfg). Lines 12 to 14 define ByteStrings that are used to hold the telegrams. Finally a oCommu instance of type ADX_toCommu is defined in line 15. This object provides basic communication functionality like connecting and disconnecting the target, error handling as well as sending and receiving messages. Usually you wouldn't use the ADX_toCommu object instead you would us other objects, that use an ADX_toCommu object internally. For simplicity (in means of number of objects to deal with) we will use this object in our first example.

Lines 21 to 23 define variables for the stack. The procedure vInitStack uses the recommended approach in that a preconfigured stack is loaded.

Line 54 initialises the oCommu instance passing the stack instance to use for communication and a format object instance. The purpose of the format object is explained later, since it is not necessary to understand its purpose right know. (For the curious, the format object instance defines the byte order for the transmission).

In lines 56 to 59 we define the read telegram that consists of a command byte ('R'), a length byte (we want to read one byte) and the address we want to read (0x1234).

In lines 61 to 65 we define the write telegram that consists of a command byte ('W'), a length byte (we want to write one byte), the address we want to write to (0x1234) and the byte we want to write.

Note: You have to comment out some lines in the tst_bskdsim.pool file in order to get the example running. The import of RDM_SYM, and the declaration and definition of the vReadTest procedure.

So let's experiment a little with the example using the Commander

// Read the byte at address 0x1234

- oCommu.boGenericMessage(bsCmdRd, bsResult);
- Writeln(bsResult);

Output: (82,0)

The resulting telegram contains the command (ASCII 82 equals 'R') and the data byte which is 0 in this case

// Write 0x33 to address 0x1234

- oCommu.boGenericMessage(bsCmdWr, bsResult);
- Writeln(bsResult);

Output: (6)

The resulting telegram contains the acknowledge code byte (ASCII 6 equals ACK)

// Read the byte at address 0x1234

- oCommu.boGenericMessage(bsCmdRd, bsResult);
- Writeln(bsResult);

Output: (82,51)

The resulting telegram contains the command and the data byte read (51 equals 0x33)

With this simple example we can read and write the memory of the ECU.

4 Telegrams

As you could see in the previous example ADX allows accessing the target via objects, although the example did not show the real benefits of accessing the target via objects. Better examples are provided later in the tutorial.

What you could see also is the layered approach. All parameters that specify the communication channel like the COM port the bit rate and so on are defined in the preconfigured RDM.aida-cfg file. The application just has to load the stack and afterwards can forget all the addressing (target) and timing stuff.

On the other hand the stack does not need to know anything about the data that is transmitted via the stack.

Fig. 2 shows the typical assembly of a telegram. The white fields are created by the application. The application passes this 'block of bytes' to the AIDA stack. The AIDA stack generates the grey fields and sends the telegram to the target. When the target sends an answer the AIDA stack stripes of the grey fields and passes a 'block of data' to the application. The application itself knows how to handle the data.

sender	receiver	length	function code	data	checksum

Figure 2: Typical telegram assembly

In the example we used byte arrays to communicate with the target, which is pretty much a block of data. In the following chapter we will introduce some ADX objects which allow a more structured view of the data and an object oriented way of accessing the data. This is the real advantage of ADX.

4.1 The NORM_toFormat object

Before we can delve into the details of ADX we have to introduce the NORM_toFormat object and its use.

When you use objects instead of bytes it is possible that you have to deal with data types that are bigger than bytes like an Int32. Different processors have a different internal representation of the Int32 data type. Fig. 3 shows on the left hand side the representation of 0xAABBCCDD (an Int32) on an Alpha PowerPC (the so called BigEndian byte order) and on the right hand side the representation of the same value on an Intel x86 machine (the so called LittleEndian byte order).

This is the reason why we imported the NORM library. The NORM library defines the NORM_toFormat object which takes care of byte order. All you have to do is to create an instance of the NORM_toFormat object (line 23 in the previous example) initialise the instance with the byte order of the target (line 51) and pass the instance to the poInit function of the ADX_toCommu instance (line 54).

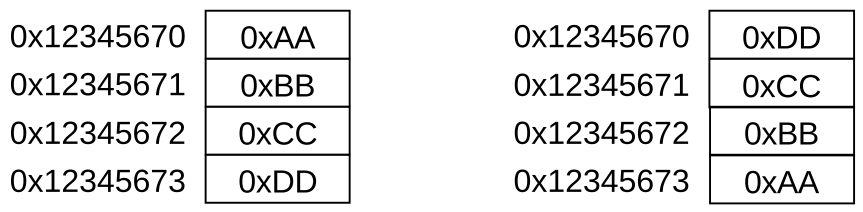

Figure 3: Byte order on an Alpha PowerPC and an Intel x86

```
Var oFormat: NORM_toFormat;
Begin
  oFormat.poInit(NORM_nenBOLittleEndian);  // or NORM_nenBOBigEndian
  ...
```

```
type
  NORM_tenByteOrder = ( NORM_nenBOUnknown, NORM_nenBOBigEndian,
                        NORM_nenBOLittleEndian {,NORM_nenBOMiddleEndian} );
  NORM_tenDirection = ( NORM_nenDirToHost, NORM_nenDirToECU);
  NORM_tenBOType    = ( NORM_nenBOTWord16, NORM_nenBOTWord32,
                        NORM_nenBOTWord64, NORM_nenBOTReal32,
                        NORM_nenBOTReal64, NORM_nenBOTPointer );
  NORM_taabsVector  = array [NORM_tenBOType] of array [NORM_tenDirection]
                      of ByteString;
  NORM_tpaabsVector = ^NORM_taabsVector;

type
  NORM_toFormat = object
    paabsVector: NORM_tpaabsVector;
    constructor poInit (enECUByteOrder: NORM_tenByteOrder);
    procedure   vTranslate(enDirection: NORM_tenDirection; pstType: tpstType;
                           pvData: Pointer; xoDOffs, xoDSize: tSize);
virtual;
    procedure   vCollectData (var bsData: ByteString; ..);
  end;
  NORM_tpoFormat = ^NORM_toFormat;
```

The vTranslate method is not interesting to you. It is only used internally by objects that use the format object.

You can use the vCollectData method to convert a list of passed parameters into a ByteString in target byte order.

Adjustments that require more than modifying the byte order e.g. different number of bits of mantissa or exponent of a real number can be done with the same mechanism. All you have to do is to derive an object from NORM_toFormat and override the vTranslate method to handle this case. The rest of the application does not need to be touched.

5 ADX objects

This chapter explains the object provided in the ADX library in detail and provides examples that show how to use the objects.

5.1 ADX_toCommu

The communication object provides basic functionalities to other ADX objects. There are methods to connect and disconnect the target, to send messages, and to handle errors. The boGenericMessage of the object can be used to send and receive data blocks of arbitrary size via the stack. Usually you wouldn't use this object in your application. Instead the object provides access methods to other objects that use the ADX_toCommu object internally. These objects are explained later in this chapter.

Taken from ADX.pli

```
{ ADX_toCommu: ECU dependent communication object }
{ ---------------------------------------------- }
{ This object provides session dependent information and methods.         }
type
  ADX_toCommu = object
    wFlags:             Word;           { flag word                         }
    bTries:             Byte;           { max. no. of tries for offline,    |
                                        | stack, negative response errors (1}
    bRLevel:            Byte;           { level of recursion                }
    poStack:            AIDA_tpoStack;  { stack object of commu channel     }
    pstParamList:       AIDA_tpstParamList; { n.u. (todo)                   }
    poFormat:           NORM_tpoFormat; { associated byte order object      }
    bsData:             ByteString;     { local data buffer                 }
    stError:            ADX_tstError;   { error informations (only for use  |
                                        | in error handlers etc.)           }
    constructor poInit (poStackPar: AIDA_tpoStack;
                        poFormatPar: NORM_tpoFormat);
    function    enConnect(boImmediate: Boolean): ADX_tenResult; virtual;
    function    enDisconnect(boImmediate: Boolean):ADX_tenResult;virtual;
    function    enErrorHandler (pstEvent: AIDA_tpstEvent): ADX_tenResult;
      virtual;
    function    enNegRspHandler(pstEvent: AIDA_tpstEvent): ADX_tenResult;
      virtual;
    function    csNegRsp2Str (dwNegRsp: DWord): String; virtual;
    function    csError2Str (var stErrorPar: ADX_tstError): String; virtual;
    { for lowest level messages only, e.g. for use in error handlers: }
    function    boGenericMessage (var bsTrData, bsRcData: ByteString):
      Boolean;
    function    boFreeMsg(..): Boolean;
  end; { ADX_toCommu }
  ADX_tpoCommu = ^ADX_toCommu;
```

The enConnect, enDisconnect, enErrorHandler, and enNegRspHandler functions must not change any application data.

5.2 ADX_toTransfer

A telegram can consist of additional information like an address (not the target address), telegram length, function code etc or just the data. The ADX_toTransfer object provides public parameters to define this additional data and to place the additional data in the right positions of the telegram.

Taken from ADX.pli

```
type
  ADX_toTransfer = object
    wFlags:     Word;           { flag word, see ADX_nTF*                  }
    dwTrCmd:    DWord;          { command code for that kind of tr. message }
    bTrDataPos: Byte;           { pos of first data byte in transmit message|
                                | has to be set even if there's no data !   }
    unTrOrder:  ADX_tunHeader;  { dedicated byte order of transmit header   }
    dwRcCmd:    DWord;          { command code for that kind of rc. message }
    bRcDataPos: Byte;           { pos of first data byte in receive message |
                                | if bRcDataPos>size of receive header then |
                                | some bytes are dropped without check      }
    unRcOrder:  ADX_tunHeader;  { dedicated byte order of receive header    }
    constructor poInit;
   private
    function    boTransfer (poCommu: ADX_tpoCommu; var bsRcData: ByteString;
                            dwSource: DWord; dwLen: DWord;
                            bsTrData: ByteString): Boolean;
  end; { ADX_toTransfer }
  ADX_tpoTransfer = ^ADX_toTransfer;
```

The object contains a byte flag parameter that indicates the transfer mode and a two parameter sets, one for sending telegrams and one for receiving.

Let's use our previous example, where the telegram consisted of a function code byte, followed by a length byte and two address bytes with the high byte first.

Example ADX_toTransfer instance

```
oMemRr: ADX_toTransfer = (
  wFlags:         ADX_nTFSegmentedRead;
  dwTrCmd:        Ord('R');
  bTrDataPos:     4;
  unTrOrder: (
    unCommand:    (abA: (0xFF, 0xFF, 0xFF, 0));
    unDataSource: (abA: (0xFF, 0xFF, 2, 3));
    unDataSize:   (abA: (0xFF, 0xFF, 0xFF, 1))
  );
  dwRCCmd:        Ord('R');
  bRcDataPos:     1;
  unRcOrder: (
    unCommand:    (abA: (0xFF, 0xFF, 0xFF, 0));
    unDataSource: (abA: (0xFF, 0xFF, 0xFF, 0xFF));
    unDataSize:   (abA: (0xFF, 0xFF, 0xFF, 0xFF))
  );
```

The wFlags parameter is set to ADX_nTFSegmentedRead i.e. read requests that want to read more bytes than the target can handle in one telegram can be split into more telegrams. dwTrCmd contains the function code (read access in this case) and bTrDataPos indicates the header size (4 bytes in our example as there are one command byte, a length byte, and two address bytes) or as its name indicates the position of the first data byte. Finally the unTrOrder parameter specifies the order of the header information. The unCommand parameter could consist of four bytes. The first three 0xFF indicates that the bytes are not used. The zero in the unCommand parameter indicates that the least significant byte of the four possible bytes is placed first in the header. The one in the unDataSize parameter indicates that the fourth length byte is placed as second byte in the header and the parameter unDataSource specifies that the third byte of the parameter is placed after the length byte followed by the fourth byte. Fig. 4 shows the resulting header. The parameter set for receiving telegrams is assembled analogous.

'R'	1	0x12	0x34

Figure 4: Header of send telegram

5.3 ADX_toAccess

The ADX_toAccess object provides data access for a single data exchange operation i.e. you should not use the function for read-modify-write operations. The object itself does not specify whether the access operation reads or writes data. Typical applications of the object are diagnosis functions like ReadID, GetStatus, and SetStatus and so on.

The object contains a pointer to the ADX_toCommu object to use and an ADX_toTransfer object instance.

Taken from ADX.pli

```
type
  ADX_toAccess = object
    poCommu:    ADX_tpoCommu;   { pointer to associated communication object}
    oTransX:    ADX_toTransfer; { data transfer object                      }
    constructor poInit (poCommuPar: ADX_tpoCommu);
    function    boRdData (var bsRcData: ByteString; dwIdent: DWord;
                          xoLen: tSize; bsTrData: ByteString): Boolean;
    function    boWrData (var bsRcData: ByteString; dwIdent: DWord;
                          xoLen: tSize; bsTrData: ByteString): Boolean;
  end; { ADX_toAccess }
  ADX_tpoAccess = ^ADX_toAccess;
```

5.4 ADX_toRdWr

The ADX_toRdWr object is used to combine multiple data access operations to a single data access operation. This is necessary for read-modify-write operations. You can use the object not only for accessing memory, but also for accessing flash, status register and so on.

Taken from ADX.pli

```
type
  ADX_toRdWr = object
    wFlags:     Word;           { flag word, see ADX_nRWF*                 }
    poCommu:    ADX_tpoCommu;   { pointer to associated communication object}
    dwAddrOffs: DWord;          { address offset for read/write operations  }
    bAddrShift: Byte;           { address shift for read/write operations   |
                                | (todo)                                    }
    dwRdSegLen: DWord;          { length of a single data read segment      }
    dwWrSegLen: DWord;          { length of a single data write segment     }
    dwRdBlkMsk: DWord;          { mask for legal read address ranges (todo) }
    dwWrBlkMsk: DWord;          { mask for legal write address ranges (todo)}
    oReadX:     ADX_toTransfer; { data read object                          }
    oWriteX:    ADX_toTransfer; { data write object                         }
    constructor poInit (poCommuPar: ADX_tpoCommu);
    function    boRdData(var bsRcData:ByteString; dwSource:DWord;
                         xoLen:tSize; var stErrorPar:ADX_tstError): Boolean;
    function    boWrData(dwDest: DWord; bsTrData: ByteString;
                         var stErrorPar:ADX_tstError): Boolean;
end; { ADX_toRdWr }
  ADX_tpoRdWr = ^ADX_toRdWr;
```

Now let's see how the objects work together.

Using the transfer and read/write objects

```
module TEST;

import AIDA;
import NORM;
import ADX;

const
  csStackName:  String = 'RDM';     {AIDA stack name}
{global variables}
var
  csStackIdent:   CharString;
  oRdWr:          ADX_toRdWr;       {ADX read/write object instance}
  bsReceiveData:  ByteString;       {received data}
  bsTransmitData: ByteString;       {transmit data}
  stError:        ADX_tstError;     {ADX error object instance}
  oCommu:         ADX_toCommu;      {ADX communication object instance}
  boResult:       Boolean;          {variable to store the return values}
```

```
private
{private variables}
var
  hStack:       tHandle;            {ECU stack handle}
  oStack:       AIDA_toStack;       {ECU stack instance}
  oFormat:      NORM_toFormat;      {ECU data format instance}

static
  oTransferRd: ADX_toTransfer = (
    wFlags:          ADX_nTFReadOnly;
    dwTrCmd:         Ord('R');
    bTrDataPos:      4;
    unTrOrder: (
      unCommand:    (abA: (0xFF, 0xFF, 0xFF, 0));
      unDataSource: (abA: (0xFF, 0xFF, 2, 3));
      unDataSize:   (abA: (0xFF, 0xFF, 0xFF, 1))
    );
    dwRcCmd:         Ord('R');
    bRcDataPos:      1;
    unRcOrder: (
      unCommand:    (abA: (0xFF, 0xFF, 0xFF, 0));
      unDataSource: (abA: (0xFF, 0xFF, 0xFF, 0xFF));
      unDataSize:   (abA: (0xFF, 0xFF, 0xFF, 0xFF))
    )
  );

static
  oTransferWr: ADX_toTransfer = (
    wFlags:          ADX_nTFWriteOnly;
    dwTrCmd:         Ord('W');
    bTrDataPos:      4;
    unTrOrder: (
      unCommand:    (abA: (0xFF, 0xFF, 0xFF, 0));
      unDataSource: (abA: (0xFF, 0xFF, 2, 3));
      unDataSize:   (abA: (0xFF, 0xFF, 0xFF, 1))
    );
    dwRcCmd:         Ord('W');
    bRcDataPos:      1;
    unRcOrder: (
      unCommand:    (abA: (0xFF, 0xFF, 0xFF, 0));
      unDataSource: (abA: (0xFF, 0xFF, 0xFF, 0xFF));
      unDataSize:   (abA: (0xFF, 0xFF, 0xFF, 0xFF))
    )
  );


{private procedures}
procedure vInitStack(var hStack: tHandle; csFilename: CharString);
var
  dwSLevelMask: DWord;
begin
  hStack := AIDA_hRestoreStack(csFilename, csStackIdent, dwSLevelMask);
  if hStack = nil then
  Writeln('Error loading stack');
  Halt(nExitSysError);
  endif;
end;


procedure vDeinit;
begin
  oRdWr.vDone;
  oCommu.vDone;
  oStack.vRemoveEvents;
  oStack.vDone;
  if hStack <> nInvHandle then
    if !AIDA_boDeleteStack(hStack) then
      Writeln("?RDM – Error deleting stack");
    endif
  endif;
end;


begin
  oFormat.poInit(NORM_nenBOLittleEndian);
  vInitStack(hStack, csStackName);
  oStack.poInit(hStack);
  oCommu.poInit(@oStack, @oFormat);
  oRdWr.poInit(@oCommu);                    {initialize read/write instance}
  oRdWr.oReadX := oTransferRd;              {set read transfer instance}
  oRdWr.oWriteX := oTransferWr;             {set write transfer instance}
  bsTransmitData := 0x33;                   {set transmit data}
end.
```

Commands executed in the Commander

```
// Read the byte at address 0x1234
oRdWr.boRdData(bsReceiveData, 0x1234, 1, stError);
Writeln(bsReceiveData);                     {prints: 0}
// Write 0x33 to address 0x1234
oRdWr.boWrData(0x1234, bsTransmitData, stError);
// Read the byte at address 0x1234
oRdWr.boRdData(bsReceiveData, 0x1234, 1, stError);
Writeln(bsReceiveData);                     {prints: 51}
```

The printing the received data to the Commander displays just the data (0 for the first call and 51 for the second call) and not the command byte. The reason is that we told the ADX_toTransfer object via the bRcDataPos parameter that the 'data' starts at the second byte that is received. Thus the first returned byte, which contains the command code is stripped of and not passed to the application.

The next subsection describes an object that allows to define an output format for the data.

5.5 ADX_toShowMsg

Usually you want to display the read and written data for diagnosis purpose. You can use the ADX_toShowMsg object to implement a function that displays the received data in an appropriate way. For example you could implement an object that displays the read data in hexadecimal format and the use this object to display data read from RAM, ROM, or EEPROM. Thus you have to implement the display function just once.

For details see the ADX.pli

Taken from ADX.pli

```
type
  ADX_toShowMsg = object
    wFlags:             Word;           { flag word, see ADX_nSF*          }
    bHdrLimit:          Byte;           { output header if bHdrLimit<>0xFF  |
                                        | Length(bsData) > bHdrLimit       }
    i8IdentDigits:      Int8;           { =0:  don't display ident as title |
                                        | <>0: display ident. with given    |
                                        |      digits (see StrfhDW) or width}
    cIdentFormat:       Char;           { format for dwIdent (sid or addr)  |
                                        | 'D': decimal                      |
                                      | ' \0','X'..: cMode for StrfhDW     }
    bWidth:             Byte;           { width for one byte               }
    bCnt:               Byte;           { no. of bytes per line (0=all in   |
                                        | one line)                        }
    bDummy:             Byte;           { reserve                          }
    csTitle:            String;         { title for data output            }
    procedure vShow(dwIdent: DWord; var bsData: ByteString);
  end; { ADX_toShowMsg }
  ADX_tpoShowMsg = ^ADX_toShowMsg;
```

An example is provided in 5.6.1

5.6 ADX_toMsg…

The ADX_toMsg object is the base object for message objects used to implement a diagnosis. The base object provides buffers for sending and receiving data, a pointer to an ADX_toShowMsg object instance and a vShow method that is declared as virtual.

Taken from ADX.pli

```
type
  ADX_toMsg = object
    bsTrData:   ByteString;
    bsRcData:   ByteString;
    stError:    ADX_tstError;   { error informations                        }
    poShow:     ADX_tpoShowMsg;
    constructor poInit (poShowPar: ADX_tpoShowMsg);
    procedure   vShow (..); virtual;
  end; { ADX_toMsg }
  ADX_tpoMsg = ^ADX_toMsg;
```

5.6.1 ADX_toMsg0

This object does not expect any parameter neither for reading nor for writing. This is the reason why '0' is attached to the object name. Both methods for reading and writing can take a list of optional parameters, allowing transmitting any data you want. A typical application is a ReadID diagnosis function and so on.

Taken from ADX.pli

```
type
  ADX_toMsg0 = object (ADX_toMsg)
    poAccess:   ADX_tpoAccess;
    constructor poInit (poAccessPar: ADX_tpoAccess; poCommuPar: ADX_tpoCommu;
                        poShowPar: ADX_tpoShowMsg);
    procedure   vShow (..); virtual;
    function    boRd (..): Boolean; ifr;
    function    boWr (..): Boolean; ifr;
  end; { ADX_toMsg0 }
  ADX_tpoMsg0 = ^ADX_toMsg0;
```

You should not use the object for read-modify-write operations.

The following example shows how to create a free telegram using the ADX_Msg0, the ADX_toAccess, and the ADX_ShowMsg objects.

Note: The example just shows excerpts from rdm.pool that are necessary to understand the use of the previously mentioned objects. The creation and initialisation of the stack and communication object was showed and explained in previous examples and is omitted in this example. See A2.1 for the complete source code of the example.

Excerpt form rdm.pool (see A2.1)

```
var
  oFree:         ADX_toMsg0;     { object for free messages                }

private

{ free messages parameter set }
static
  oFreeAccess: ADX_toAccess = (
    poCommu: nil;
    oTransX: (
      wFlags : 0;
      dwTrCmd: 0;  { n.u. }
      bTrDataPos: 0;
      unTrOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      );
      dwRcCmd: 0;  { n.u. }
      bRcDataPos: 1;  { always remove command byte }
      unRcOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      )
    )
  );

static
  oFreeShow: ADX_toShowMsg = (
    wFlags:             ADX_nSMFAscii7 or ADX_nSMFDispNoData;
    bHdrLimit:          7;
    i8IdentDigits:      0;
    cIdentFormat:       '\0';
    bWidth:             3;
    bCnt:               16;
    bDummy:             0;
    csTitle:            'Data: '
  );

{ vDeinit: Module deinit }
{ --------------------- }
procedure vDeinit;
begin
  { module clean up }
  oFree.vDone;
  ...
end; { vDeinit }


{ Module init }
{ ----------- }
begin
  ...
  { initialising message object for free message operations }
  oFree.poInit (@oFreeAccess, @oCommu, @oFreeShow);
end. { Init }
```

First oFree is created. Afterward an instance of ADX_toAccess oFreeAccess is created which provides the read/write implementation to oFree. Finally oFreeShow is created which is of type ADX_toShow. This object instance is passed to oFree as parameter in the poInit constructor and used to display the read bytes in the Commander.

So let's experiment a little with the example using the Commander.

Commands executed in the Commander

```
// Read the byte at address 0x1234
oFree.boRd('R', 1, 0x1234);
oFree.vShow;                                 {prints: 0}
// Write 0x33 to address 0x1234
```

```
oFree.boWr('W', 1, 0x1234, 0x33);
// Read the byte at address 0x1234
oFree.boRd('R', 1, 0x1234);
Writeln(bsReceiveData);                      {prints: 51}
```

As you can it is possible to create message that are intuitive to use if you know the commands that the target understands.

5.6.2 ADX_toMsg1

This object inherits from ADX_toMsg0, but expects one parameter for reading resp. writing. This is the reason why '1' is attached to the object name. Both methods for reading and writing take a DWord parameter and a list of optional parameters. A typical application is the use as status function whereby the desired status function is defined by dwIdent.

Taken from ADX.pli

```
type
  ADX_toMsg1 = object (ADX_toMsg0)
    dwLastIdent:    DWord;
    procedure   vShow (..); virtual;
    function    boRd (dwIdent: DWord; ..): Boolean; ifr;
    function    boWr (dwIdent: DWord; ..): Boolean; ifr;
  end; { ADX_toMsg1 }
  ADX_tpoMsg1 = ^ADX_toMsg1;
```

You should not use the object for read-modify-write operations.

The use of the ADX_toMsg1 object is analogous to the use of the ADX_toMsg0 object. See A2.1 for a coding example.

5.6.3 ADX_toMsg2

This object inherits from ADX_toMsg, but expects two parameters for the read method and one parameter for the write message. The first parameter of the read method is used to specify the address to read from and the second parameter is used to specify the number of bytes to read.

Taken from ADX.pli

```
type
  ADX_toMsg2 = object (ADX_toMsg)
    poRdWr:     ADX_tpoRdWr;
    dwLastAddr: DWord;
    constructor poInit (poRdWrPar: ADX_tpoRdWr; poCommuPar: ADX_tpoCommu;
                        poShowPar: ADX_tpoShowMsg);
    procedure   vShow (..); virtual;
    function    boRd (dwAddr: DWord; xoLen: tSize): Boolean; ifr;
    function    boWr (dwAddr: DWord; ..): Boolean; ifr;
  end; { ADX_toMsg2 }
  ADX_tpoMsg2 = ^ADX_toMsg2;
```

The address of the last data access is stored in dwLastAddr. You can use the value in your vShow method.

You used the ADX_toMsg2 object during the setup test. See A2.1 for a coding example.

5.7 Overview diagram of the ADX objects

Fig. 5 shows the objects and their relationship.

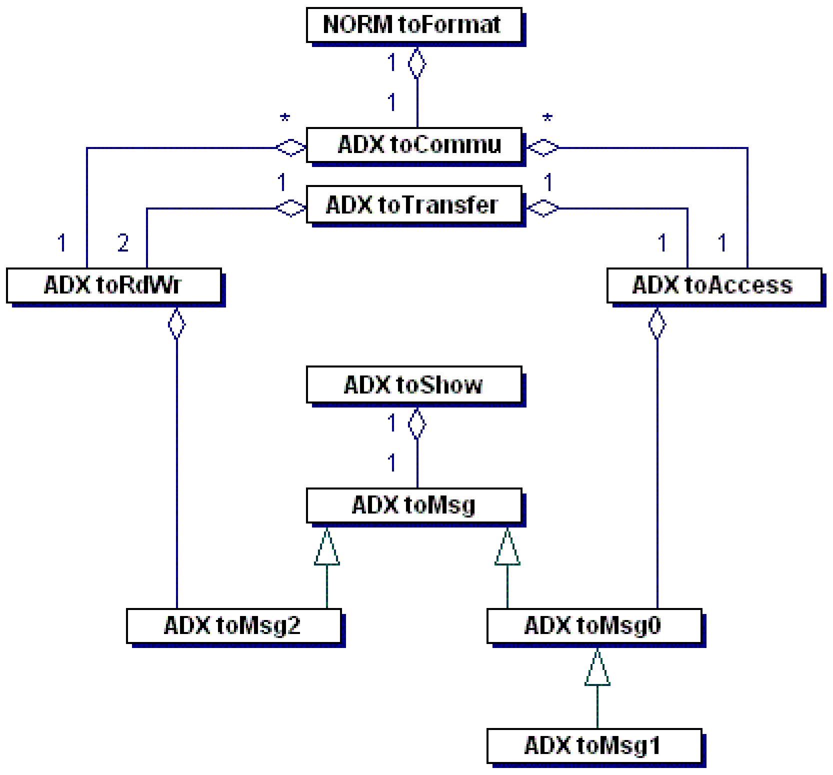

Figure 5: ADX objects and their relationship

6 Symbol variables

Variables of the target program and their memory locations are listed in a symbol file after the source code is compiled and linked. Each entry consists at least of a name and a value and optional additional type information.

6.1 Data representation in a diagnosis program

A Symbol generator creates a .pool file that contains the variables of the target in POOL data types and notation, if the symbol tables (or cross-reference lists) contains the needed information. The following shows an example conversion of the ODO module (in C) of a control device into POOL.

```
word  ODO_wKFactor     à  ODO_wKFactor:   Word
byte  ODO_bFlags       à  ODO_bFlags:     Byte
int16 ODO_i16Counter   à  ODO_i16Counter: Int16
```

POOL has defined compatible data types for all (or at least almost all) base types of control devices. This allows accessing the variables in the diagnosis program using the same symbolic name as is used in the source code of the control device. Further more the range of the variables is in POOL the same as in C.

6.2 ADX_toVar

The ACX_toVar object describes the properties of a control device variable containing all information needed to exchange the variable's content between the PC and the control device (ECU).

ADX_toVar

```
type
  ADX_toVar = object                    { object type for ECU variables     }
    pabData:            tpaByte;        { pointer to data image             }
    pstType:            tpstType;       { pointer to data type structure    }
    poRdWr:             ADX_tpoRdWr;    { pointer to data rd/wr object      }
    dwSource:           DWord;          { data source address within ECU    }
    stError:            ADX_tstError;   { error informations                }
    {dwTimeStamp:       DWord;}         { time stamp of last refresh (todo?)}
    { initialisation }
    constructor poInit (poRdWrPar: ADX_tpoRdWr; dwAddress: DWord; var xData);
    { read/write functions for the entire data structure }
    function    boRd:  Boolean; { implicite read of entire variable data    }
    function    boWr:  Boolean; { implicite write of entire variable data   }
    { read/write functions for partial read/writes of the data structure }
    function    boRdField   (var xParam): Boolean;
    function    boWrField   (var xParam): Boolean;
    { display entire data structure }
    procedure   vShow (..); virtual;
    { read access to local data in variant sections considering the endianess
}
    function    i16Get      (var abParam: tabInt16):   Int16;
    function    i32Get      (var abParam: tabInt32):   Int32;
    function    wGet        (var abParam: tabWord):    Word;
    function    dwGet       (var abParam: tabDWord):   DWord;
    function    r64Get      (var abParam: tabReal64):  Real64;
    function    r32Get      (var abParam: tabReal32):  Real32;
    function    pGet        (var abParam: tabPointer): Pointer;
    { write access to local data in variant sections considering the
endianess }
    procedure   vPutInt16   (var abParam: tabInt16;    i16Value: Int16);
    procedure   vPutInt32   (var abParam: tabInt32;    i32Value: Int32);
    procedure   vPutWord    (var abParam: tabWord;     wValue:   Word);
    procedure   vPutDWord   (var abParam: tabDWord;    dwValue:  DWord);
    procedure   vPutReal64  (var abParam: tabReal64;   r64Value: Real64);
    procedure   vPutReal32  (var abParam: tabReal32;   r32Value: Real32);
    procedure   vPutPointer (var abParam: tabPointer;  pValue:   Pointer);
  end; { ADX_toVar }
  ADX_tpoVar = ^ADX_toVar;
```

The pabData parameter of the object contains a pointer to the local copy of the control device variable. pstType points to a block describing the data type of the variable. poRdWr points to the ADX_toRdWr object, which is used to read resp. write the variable. Finally the dwSource parameter holds the address of the variable in the memory of the control device. The functions boRd and boWr are used to read/write variables from/to the control device.

6.3 Implementation of symbol variables

The implementation of symbol variables requires a lot of typing and should be done be a symbol generator tool (which does not exist currently but will be provided in the future) if more than just a couple of variables should be accessed. The variables should be defined in a separate module which has to be imported by the diagnosis application. The file should be named "MyECU_Sym.pool".

Each variable requires three steps>

1. declaration of the variable
2. declaration of the according description object
3. initialisation (and deinitialisation) of the description object

```

var                             {step 1: variable declaration}
  ODO_wKFactor:   Word;
  ODO_bFlags:     Byte;
  ODO_i16Counter: Int16;

```

```
var                                {step 2: description object declaration}
  o_ODO_wKFactor:   ADX_toVar;
  o_ODO_bFlags:     ADX_toVar;
  o_ODO_i16Counter: ADX_toVar;

  ...

{vDeinit: module deinit}
procedure vDeinit;                 {step 3b: deinitialisation of variables}
begin
  o_ODO_wKFactor.vDone;
  o_ODO_bFlags.vDone;
  o_ODO_i16Counter.vDone;
end;

{module init}
begin                              {step 3a: initialization of variables}
  o_ODO_wKFactor.poInit(@RDM.oRWMBA, 0x1234, ODO_wKFactor);
  o_ODO_bFlags.poInit(@RDM.oRWMBA, 0x2345, ODO_bFlags);
  o_ODO_i16Counter.poInit(@RDM.oRWMBA, 0xFFB8, ODO_i16Counter);
end.
```

Please notice the nomenclature for the description objects. The recommended way is to prepend the module name to the variable name and to name the description object like the described object prepending a leading 'o_'. This ensures a mapping between the variable object and the corresponding description object.

Please see the next subsection for an example on how to use symbolic variables.

6.4 Structured symbol variables

The use of structured symbol variables is analogous to the use of unstructured symbol variables. The sole exception is that the symbol generator tool (or the programmer) has to create a structure that matches the desired structure of the control device. The other steps like declaration of the variable and the belonging description object as well as the initialisation are the same.

```
type
  ODO_tenTripStat = (ODO_nen_TSInvalid, ODO_nen_TSReset, ODO_nen_TSValid);
  ODO_tstTripStruct = record
    bFlags:   Byte;
    dwValue:  DWord;
    pstPtr:   ^ODO_tstVarBuffer;
    adwValue: array[0..3] of DWord;
    enStat:   ODO_tenTripStat;
  end;

  ODO_tstVarBuffer = record
    Variant
      (ab_i32Value: tabInt32);
      (ab_dwValue:  tabDWord);
      (ab_rValue:   tabReal32);
  end;

var
  ODO_stTripStruct:    ODO_tstTripStruct;
  ODO_stVarBuffer:     ODO_tstVarBuffer;

  O_ODO_stTripStruct:  ADX_toVar;
  O_ODO_stVarBuffer:   ADX_toVar;

  ...

  {vDeinit: module deinit}
  procedure vDeinit;
  begin
    o_ODO_stTripStruct.vDone;
    o_ODO_stVarBuffer.vDone;
  end;

  {module init}
  begin
    o_ODO_stTripStruct.poInit(@RDM.oRWMBA, 0x1234, ODO_stTripStruct);
    o_ODO_stVarBuffer.poInit(@RDM.oRWMBA, 0x1234, ODO_stVarBuffer);
  end.
```

Sometimes you have to work with the fields of structures, contrary to the unstructured symbol variables where you always work on the whole variable. The ADX_toVar object provides several methods to address the fields. Further more there are several …Get, vPut… methods, which allow to access the fields even if there is no type conversion (e.g. with C unions) possible. The Get and vPut methods access only local copies and perform no explicit read or write operation on the control device.

Commands executed in the Commander

```
// Read target data and store data in the local stTripStruct structure
o_ODO_stTripStruct.boRd
// Display the read value
o_ODO_stTripStruct.vShow
{prints: bFlags:0;dwValue:0;pstPtr:@:00000000;adwValue(0,0,0,0);enStat:
         ODO_nenTSInvalid)}
// Write some values to the stTripStruct structure
Writeln(ODO_stTripStruct.bFlags)              {prints: 0}
ODO_stTripStruct.bFlags := 0xAA               {write to the local copy}
Writeln(ODO_stTripStruct.bFlags)              {prints: 255}
ODO_stTripStruct.dwValue := 0x12345678
ODO_stTripStruct.adwValue[0] := 0x11111111
ODO_stTripStruct.adwValue[1] := 0x22222222
ODO_stTripStruct.adwValue[2] := 0x33333333
ODO_stTripStruct.adwValue[3] := 0x44444444
ODO_stTripStruct.enStat
// Write the modified stTripStruct structure to the target
o_ODO_stTripStruct.boWr
// Read the written data using oMem
oMem.boRd(0x1234,0x20)
// Display the read value
```

```
oMem.vShow

// displayed output
       4  5  6  7  8  9  A  B  C  D  E  F  0  1  2  3
1234: FF 78 56 34 12 00 00 00 00 11 11 11 11 22 22 22 ".xV4.........""""
1244: 22 33 33 33 33 44 44 44 44 02 00 00 00 00 00 00 ""3333DDDD......."
```

As you can see in this example ADX allows to access the variables as objects using the names of their C counterparts (of course with the o_ODO prefix). This is the real power of ADX. You could also use the o_ODO_stTripStruct instance to reread and verify that the previous write access was successful. We did use the oMem object to verify that the data was written correctly since the o_ODO_stTripStruct does not implement a vShow method so verification is easier using the oMem instance in this case. Of course there is still one weakness of the current approach. When the C code is compiled again, the addresses could change. Till now we used hardcoded addresses. The next chapter explains how to use the symbol file to gain the addresses during runtime, so that you do not have to modify your diagnosis program each time you compiled your C code.

6.5 Using symbol files

The following example is an excerpt that shows how to use a symbol file. For the complete source code see A2.3.
Note: You can not test the following example since this example requires a real target. This example is intendet to show how to implement diagnosis functions using ADX and a symbol file.

TNC_Sym.pool

```
// Structur TNC__tstData      {definition of the structure}
type
  TNC__tstData = record
     bState:       Byte;      // status
     bInjFuel:     Byte;      // consumed fuel (injected fuel)
     biLowDamp:    Byte;      // low damp / refill detection
     biRemote:     Byte;      // remote (at overfill)
     biOverfill:   Byte;      // tank overfilled
     biDeductOpt:  Byte;      // Optional (depends on compiler switch -
                              // position is always occupied with one byte)
     biOnToActOpt: Byte;      // Optional not always existing
     bIncState:    Byte;      // status
     bDamp_Cnt:    Byte;      // -JJ-  10.Jan.02   Added for Lin Damp */

// define additional types here

end;
```

The file above defines the structure TNC__tstData which is used in the C code.

TNC.pool

```
// Declaration of symbol variables used in diagnosis object
// (the variables have the same name and datatype as the variable at the ecu)
var                                       {step 1}
  TNC__stData:         TNC__tstData;
  ...

var                                       {step 2}
  o_TNC__stData:       ADX_toVar;
  ...

{vDeinit: module deinit}                  {step 3b}
procedure vDeinit;
begin
  oSymList.vDone;                            // Call destructor of sym-List

  // Close symbol file
  if !TClose(fiSymFile) then
    Writeln(GetErrorMsg(GetError));
    return;
  endif;

  // Deinit of var-objects
  o_TNC__stData.vDone;
  ...
end;

{module init}                             {step 3a}
begin
  // Get and check SW-ID
  if (oID.csGetSwNo = "01 50") then
    csSymFileName := "../bin/sym/Toyota.sym";
  // Add further symbol files
  // elseif (oID.csGetSwNo = ".....") then
  else                                         // Happens also if Com-Port
                                               // isn't available
    Writeln("No valid sw-number");
    return;
  endif;

  // Open symbol file
  if !TOpen(fiSymFile,csSymFileName,nenFMRead) then  // Open sym file
    Writeln(GetErrorMsg(GetError));
    return;
  endif;

  // Init of symbol list
  if oSymList.poInit = nil then
    Writeln(GetErrorMsg(GetError));
    return;
  endif;

  // Set paramter for read symbol
  oSymList.vSetRdPar({wFlagsPar:}BDSYM_nBFReplDup|BDSYM_nBFIgnSynErr|
                   BDSYM_nBFIgnExprErr,{pstBitAddrDef:} nil,
                   {dwInvAddrPar:} 0);
  // Read Sym.File into Sym. List
  if(oSymList.enRead(fiSymFile) <> nenELSuccess) then
    oSymList.vDone;
    TClose(fiSymFile);
    Writeln("Read sym. File failed.");
    return;
  endif;

  // Init. of var - objects (for details see description of ADX-Library)
  // Parameters:
  // @oMemRdWrAccess:
  //             Read/Write object from ASSP3_IF
  //
  // oSymList.dwGetAddr("TNI_wFuelVolume",true):
  //             Reads the address from TNI_wFuelVolume, from the Sym. List
  //
  // TNC_strData
  //            Symb. variable declared in this file

  o_TNC__stData.poInit(@oMemRdWrAccess,oSymList.dwGetAddr("TNC__stData",
                      true), TNC__stData);
end.
```

The difference between this example and the previous example is that this example loads a symbol file which contains the symbols and their respective addresses. Notice that the function call to the o_TNC__stData.poInit function in this example uses the oSymList.dwGetAddr function of the BDSYM library to resolve the address whereas the o_ODO_stTripStruct.poInit function call in the previous example used a hardcoded address.

Conclusion:

The previously techniques showed how to implement a diagnosis application using ADX that allows to access the variables of the target via their name without any modification of the diagnosis application even if the C source code is compliled again.

New variables would need a modification of the diagnosis example of course.

Annex

A1 Bibliography

Tutorials:

POOL-Tutorial part 1 – Basics of the POOL programming language, by T.Locker

Revised by U. Kühn.

Simple introduction to the basics of the POOL programming language, inclusive lots of examples and exercises.

POOL-Tutorial part 2 – OOP and POOL, by T.Locker

Revised by U. Kühn.

Comprehendible introduction to object oriented programming using POOL, inclusive lots of examples and exercises.

POOL-Tutorial part 3 – POOL libraries, by T.Locker

Revised by U. Kühn.

Comprehendible introduction to the POOL library functions, inclusive lots of examples.

A2 Source code

A2.1 rdm.pool

```
{============================================================================
  RDM:
  ECU interface setup for sample RDM project
  using BSK diagnostic standard
  (c) 2001-2002 BSK Datentechnik GmbH

  Authors:    Uwe Kuehn,      BSK Datentechnik GmbH
              Hans Schmidts,  BSK Datentechnik GmbH

  $Revision: 1.3 $
  $Date: 2002/10/30 09:48:53 $

  Modification history:
  2002-09-23 BSK/HS New names *GetErrorMsg and nExit*
  2001-11-27 BSK/HS Completely revised
  2001-10-19 BSK/UK First release
  2001-09-14 BSK/HS Comments added
  2001-07-13 BSK/HS Comments added
  2001-06-29 BSK/HS CVS tags added
  2001-05-29 BSK/UK First version

============================================================================}

module RDM;

import AIDA;
import NORM;
import ADX;


{ csStackName: Name of used AIDA stack }
{ ------------------------------------ }
{ (uses environment variables RDM_COM_PORT and RDM_BDIAG_ECHOSUPPRESSION) }
const
  csStackName: String = 'RDM';
var
  csStackIdent:  CharString;


{ toID: Deviated message object with special show method }
{ ------------------------------------------------------ }
type
  toID = object (ADX_toMsg0)
    procedure vShow (..); virtual;
  end;

var
  oFree:         ADX_toMsg0;     { object for free messages                }
  oID:           toID;           { ECU adapted object for ID access         }
  oMem:          ADX_toMsg2;     { object for memory access                 }
  oEEP:          ADX_toMsg2;     { object for non volatile memory access    }
  oSR:           ADX_toMsg1;     { object for status read operations        }
  oSW:           ADX_toMsg1;     { object for status write operations       }
  oDE:           ADX_toMsg1;     { object for end diagnostics messages      }


{=============================================================================}
private

{ RDM__DBG_DumpExec: Dump execution }
{ --------------------------------- }
{ $define RDM__DBG_DumpExec}


{ free messages parameter set }
static
  oFreeAccess: ADX_toAccess = (
    poCommu: nil;
    oTransX: (
      wFlags : 0;
      dwTrCmd: 0;  { n.u. }
      bTrDataPos: 0;
      unTrOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      );
      dwRcCmd: 0;  { n.u. }
      bRcDataPos: 1;  { always remove command byte }
      unRcOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      )
    )
  );

static
  oFreeShow: ADX_toShowMsg = (
    wFlags:             ADX_nSMFAscii7 or ADX_nSMFDispNoData;
    bHdrLimit:          7;
    i8IdentDigits:      0;
    cIdentFormat:       '\0';
    bWidth:             3;
    bCnt:               16;
    bDummy:             0;
    csTitle:            'Data: '
  );


{ read identifier parameter set }
static
  oIDAccess: ADX_toAccess = (
    poCommu: nil;
    oTransX: (
      wFlags : ADX_nTFReadOnly;
      dwTrCmd: Ord('I');
      bTrDataPos: 1;
      unTrOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      );
      dwRcCmd: Ord('I');
      bRcDataPos: 1;
      unRcOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      )
    )
  );

static
  oIDShow: ADX_toShowMsg = (
    wFlags:             ADX_nSMFDispNoData;
    bHdrLimit:          7;
    i8IdentDigits:      0;
    cIdentFormat:       '\0';
    bWidth:             3;
    bCnt:               16;
    bDummy:             0;
    csTitle:            'ID: '
  );



{ read/write memory by address parameter set }
static
  oMemRdWr: ADX_toRdWr = (
    wFlags : ADX_nRWFSegmentedRead or ADX_nRWFSegmentedWrite;
    poCommu: nil;
    dwAddrOffs: 0;
    bAddrShift: 0;
    dwRdSegLen: 8;
    dwWrSegLen: 4;
    dwRdBlkMsk: 0xFFFFFFFF;
    dwWrBlkMsk: 0xFFFFFFFF;
    oReadX: (
      wFlags : 0;
      dwTrCmd: Ord('R');
      bTrDataPos: 4;
      unTrOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,   2,   3));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,   1))
      );
      dwRcCmd: Ord('R');
      bRcDataPos: 1;
      unRcOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      )
    );
    oWriteX: (
      wFlags : 0;
      dwTrCmd: Ord('W');
      bTrDataPos: 4;
      unTrOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,   2,   3));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,   1))
      );
      dwRcCmd: Ord(ncACK);
      bRcDataPos: 1;
      unRcOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      )
    )
  );

static
  oMemEEPShow: ADX_toShowMsg = (
    wFlags:              ADX_nSMFAscii7 or ADX_nSMFDispNoData or
ADX_nSMFRepeatIdent;
    bHdrLimit:           7;
    i8IdentDigits:       4;
    cIdentFormat:        '\0';
    bWidth:              3;
    bCnt:                16;
    bDummy:              0;
    csTitle:             ''
  );



{ read/write non-volatile memory parameter set }
static
  oEEPRdWr: ADX_toRdWr = (
    wFlags : ADX_nRWFSegmentedRead or ADX_nRWFSegmentedWrite;
    poCommu: nil;
    dwAddrOffs: 0;
    bAddrShift: 0;
    dwRdSegLen: 8;
    dwWrSegLen: 4;
    dwRdBlkMsk: 0xFFFFFFFF;
    dwWrBlkMsk: 0xFFFFFFFF;
    oReadX: (
      wFlags : 0;
      dwTrCmd: Ord('r');
      bTrDataPos: 4;
      unTrOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,   2,   3));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,   1))
      );
      dwRcCmd: Ord('r');
      bRcDataPos: 1;
      unRcOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      )
    );
    oWriteX: (
      wFlags : 0;
      dwTrCmd: Ord('w');
      bTrDataPos: 4;
      unTrOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,   2,   3));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,   1))
      );
      dwRcCmd: Ord(ncACK);
      bRcDataPos: 1;
      unRcOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      )
    )
  );


{ read status parameter set }
static
  oSRAccess: ADX_toAccess = (
    poCommu: nil;
    oTransX: (
      wFlags : ADX_nTFReadOnly;
      dwTrCmd: Ord('G'); { Get }
      bTrDataPos: 2;
      unTrOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,   1));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      );
      dwRcCmd: Ord('G');
      bRcDataPos: 1;
      unRcOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      )
    )
  );

static
  oSRShow: ADX_toShowMsg = (
    wFlags:             ADX_nSMFAscii7 or ADX_nSMFDispNoData;
    bHdrLimit:          7;
    i8IdentDigits:      2;
    cIdentFormat:       '\0';
    bWidth:             3;
    bCnt:               16;
    bDummy:             0;
    csTitle:            'Status '
  );

{ set status parameter set }
static
  oSWAccess: ADX_toAccess = (
    poCommu: nil;
    oTransX: (
      wFlags : ADX_nTFWriteOnly;
      dwTrCmd: Ord('S'); { Set }
      bTrDataPos: 2;
      unTrOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,   1));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      );
      dwRcCmd: Ord(ncACK);
      bRcDataPos: 1;
      unRcOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      )
    )
  );

static
  oSWShow: ADX_toShowMsg = (
    wFlags:             ADX_nSMFAscii7 or ADX_nSMFDispNoData;
    bHdrLimit:          7;
    i8IdentDigits:      2;
    cIdentFormat:       '\0';
    bWidth:             3;
    bCnt:               16;
    bDummy:             0;
    csTitle:            'Status write '
  );


{ diag end parameter set }
static
  oDEAccess: ADX_toAccess = (
    poCommu: nil;
    oTransX: (
      wFlags : ADX_nTFWriteOnly;
      dwTrCmd: Ord(ncETB);
      bTrDataPos: 2;
      unTrOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,   1));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      );
      dwRcCmd: Ord(ncACK);
      bRcDataPos: 1;
      unRcOrder: (
        unCommand:    (abA: (0xFF,0xFF,0xFF,   0));
        unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
        unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
      )
    )
  );

static
  oDEShow: ADX_toShowMsg = (
    wFlags:             ADX_nSMFAscii7 or ADX_nSMFDispNoData;
    bHdrLimit:          7;
    i8IdentDigits:      2;
    cIdentFormat:       '\0';
    bWidth:             3;
    bCnt:               16;
    bDummy:             0;
    csTitle:            'Diag end '
  );


{ Private variables }
{ ================= }
var
  hStack:        tHandle;       { ECU stack handle                          }
  oStack:        AIDA_toStack;  { ECU stack object                          }
  oCommu:        ADX_toCommu;   { ECU communication object                  }
  oFormat:       NORM_toFormat; { ECU data format object                    }


{ Local Objects }
{ ============= }
procedure toID.vShow (..);
begin
  if oSelf.stError.enError<>ADX_nenEOk then
    inherited vShow(pstVAStart);
  else
    Writeln ('    SWNo HWNo  DD   MM   YY');
    Writeln ('ID:  ', Bin2HexD(bsRcData,'   '));
  endif;
end;


{ Local Procedures }
{ ================ }
procedure vInitStack (var hStack: tHandle; csFileName: CharString);
var
  dwSLevelMask:  DWord;
begin
  hStack := AIDA_hRestoreStack (csFileName, csStackIdent, dwSLevelMask);
  if hStack = nil then
    Writeln ('?RDM - Error Loading Stack');
    Writeln (AIDA_csGetErrorMsg(GetError));
    Halt(nExitSysError);
  endif;
end; { vInitStack }


{ vDeinit: Module deinit }
{ ---------------------- }
procedure vDeinit;
begin
```

```
  { module clean up }
  oDE.vDone;
  oSW.vDone;
  oSR.vDone;
  oEEP.vDone;
  oMem.vDone;
  oID.vDone;
  oFree.vDone;

  oCommu.vDone;
  oStack.vRemoveEvents;
  oStack.vDone;
  oFormat.vDone;

  if hStack <> nInvHandle then
    if !AIDA_boDeleteStack (hStack) then
      Writeln ('?RDM - Error deleting stack.');
    endif;
  endif;
end; { vDeinit }


{ Module init }
{ ----------- }
begin
  { Interface Setup for RDM ECU }
  { ---------------------------- }
  { setting up data access format for RDM ECU }
  oFormat.poInit (NORM_nenBOLittleEndian);
  { loading driver stack }
  vInitStack (hStack, csStackName);
  oStack.poInit (hStack);
  { initialising communication object }
  oCommu.poInit (@oStack, @oFormat);

  { Pre-defined message objects }
  { --------------------------- }
  { initialising message object for free message operations }
  oFree.poInit (@oFreeAccess, @oCommu, @oFreeShow);
  { initialising message object for ID read operations }
  oID.poInit (@oIDAccess, @oCommu, @oIDShow);
  { initialising message object for memory read and write operations }
  oMem.poInit (@oMemRdWr, @oCommu, @oMemEEPShow);
  { initialising message object for non-volatile memory read and write
    operations }
  oEEP.poInit (@oEEPRdWr, @oCommu, @oMemEEPShow);
  { initialising message object for read status operations }
  oSR.poInit (@oSRAccess, @oCommu, @oSRShow);
  { initialising message object for write status operations }
  oSW.poInit (@oSWAccess, @oCommu, @oSWShow);
  { initialising message object for end diagnostics operations }
  oDE.poInit (@oDEAccess, @oCommu, @oDEShow);
end. { Init }

{ eof rdm.pool }
```

A2.2 rdm_sym.pool

```
{==============================================================================

  RDM_SYM:
  Test structures for basic diagnostic functions
  (c) 2001 BSK Datentechnik GmbH

  Authors:   Uwe Kuehn,      BSK Datentechnik GmbH
             Hans Schmidts,  BSK Datentechnik GmbH

  $Revision: 1.2 $
  $Date: 2001/12/04 10:22:13 $

  Modification history:
  2001-11-23 BSK/HS Revised
  2001-10-19 BSK/UK First release
  2001-02-13 BSK/UK First revision

==============================================================================}

module RDM_SYM;

import ADX;
import RDM;


{ ODO variable types }
{ ================== }
type
  ODO_tenTripStat = (ODO_nenTSInvalid,ODO_nenTSReset,ODO_nenTSValid);
  ODO_tstTripStruct = record
    bFlags:    Byte;
    dwValue:   DWord;
    pstPtr:    ^ODO_tstVarBuffer;
    adwValue: array[0..3] of DWord;
    enStat:    ODO_tenTripStat;
  end;

  ODO_tstVarBuffer = record
    variant
      (ab_i32Value: tabInt32);
      (ab_dwValue:  tabDWord);
      (ab_rValue:   tabReal32);
  end;


{ ODO variables }
{ ============= }
var
  ODO_stTripStruct:    ODO_tstTripStruct;
  ODO_stVarBuffer:     ODO_tstVarBuffer;
  ODO_wKFactor:        Word;

  o_ODO_stTripStruct:  ADX_toVar;
  o_ODO_stVarBuffer:   ADX_toVar;
  o_ODO_wKFactor:      ADX_toVar;
```

```
{===========================================================================}
private


{ vDeinit: Module deinit }
{ ---------------------- }
procedure vDeinit;
begin
  o_ODO_stTripStruct.vDone;
  o_ODO_stVarBuffer.vDone;
  o_ODO_wKFactor.vDone;
end; { vDeinit }


{ Module init }
{ ----------- }
begin
  o_ODO_stTripStruct.poInit  (RDM.oMem.poRdWr, 0x1234, ODO_stTripStruct);
  o_ODO_stVarBuffer.poInit   (RDM.oMem.poRdWr, 0x1234, ODO_stVarBuffer);
  o_ODO_wKFactor.poInit      (RDM.oMem.poRdWr, 0x1234, ODO_wKFactor);
end. { Init }

{ eof rdm_sym.pool }
```

A2.3 Toyota_sym.pool

```
{============================================================================

  Toyota_sym:
  Deklaration of ADX_toVar Objects from Symfile
  (c) 2002 SiemensVDO

  Author:   Thomas Locker

  $Revision: 0.9 $
  $Date: 2003/01/21

  Modification history:
  2003-01-21 T. Locker draft

============================================================================}

module TOYOTA_SYM;

 import ADX;
 import BDSYM;
 import ASSP3_IF from "../bin/ASSP3_IF.pi";  // Import basic diagn. functions
                                             // e.g. MemRead,MemWrite
 import DUTILS from "../bin/DUtils";
//**************************************************************************/
// Declaration of ECU-Variables
// Every variable used in an application needs an own ADX_toVar object, which
// needs a variable with the same name and datatype as the variable at the
// ECU Structur must be declared and can then be used like every other
// variable.
//**************************************************************************/

// Structur TND_tstData
type
  TND__tstData = record
     bState:            Byte;  // Status         /* status */
     bWarn_Cnt:         Byte;  // warning counter
     bDamp_Cnt:         Byte;  // damping counter
end;

// Structur TNC__tstData
type
  TNC__tstData = record
     bState:        Byte;       // status
     bInjFuel:      Byte;       // consumed fuel (injected fuel)
     biLowDamp:     Byte;       // low damp / refill detection
     biRemote:      Byte;       // remote (at overfill)
     biOverfill:    Byte;       // tank overfilled
     biDeductOpt:   Byte;       // Optional (depends on compiler switch -
                                // position is always occupied with one byte)
     biOnToActOpt: Byte;        // Optional not always existing
     bIncState:     Byte;       // status
     bDamp_Cnt:     Byte;       // -JJ-  10.Jan.02   Added for Lin Damp */
end;
```

```
// Structur TNC__tstRef
type
  TNC__tstRef = record
   dwVolume: DWord;           // dynamic reference volume
   bMode:    Byte;            // current modus
   bDelay:   Byte;            // ignition/stabilisation delay
   bMeanCnt: Byte;            // counter mean filter
   biValid:  Byte;            // valid flag
end;

// Structur TNC__tstRefCmp
type
  TNC__tstRefCmp = record
   dwVolume: DWord;           // dynamic reference volume
   bMode:    Byte;            // current modus
   bMeanCnt: Byte;            // counter mean filter
end;

const
  MMC_nFuelGauge = (1);

// Declaration of symbol variables used in diagnosis object
// (the variables have the same name and datatype as the variable at the ecu)
var
  TNI_wFuelVolume:     Word;
  TNI__bStat:          Byte;
  TNI__bError_Cnt:     Byte;
  TNI_biErr:           Byte;
  TND_wFuelVolume:     Word;
  TND__stData:         TND__tstData;
  TNC__stData:         TNC__tstData;
  TNC__stRef:          TNC__tstRef;
  TNC__stRefCmp:       TNC__tstRefCmp;
  SMA__Sollposition:   array[0..3] of Word;
  TNC__bActiveCnt:     Byte;

//**************************************************************************/
// Declaration of corresponding variable objects (see description of ADX for
// details)
//**************************************************************************/
var
  o_TNI_wFuelVolume:   ADX_toVar;
  o_TNI__bStat:        ADX_toVar;
  o_TNI__bError_Cnt:   ADX_toVar;
  o_TNI_biErr:         DUtils_toBitVar;

  o_TND_wFuelVolume:   ADX_toVar;
  o_TND__stData:       ADX_toVar;
  o_SMA__Sollposition: ADX_toVar;

  o_TNC__stData:       ADX_toVar;
  o_TNC__stRef:        ADX_toVar;
  o_TNC__stRefCmp:     ADX_toVar;
  o_TNC__bActiveCnt:   ADX_toVar;
```

```
private
//***********************************************************************/
// Variable for the symbol file
//***********************************************************************/
var
  oSymList:           BDSYM_toBase;         // Listobject for symFile
  fiSymFile:          tFile;                // Symbolfile
  csSymFileName:      String;               // Name of Symbolfile


//***********************************************************************/
// Deinitialisation
//***********************************************************************/
procedure vDeinit;
begin
  oSymList.vDone;                           // Call destructor of sym-List

  // Close symbol file
  if !TClose(fiSymFile) then
    Writeln(GetErrorMsg(GetError));
    return;
  endif;

  // Deinit of var-objects
  o_TNI_wFuelVolume.vDone;
  o_TNI__bStat.vDone;
  o_TNI__bError_Cnt.vDone;
  o_TNI_biErr.vDone;

  o_TND_wFuelVolume.vDone;
  o_TND__stData.vDone;
  o_SMA__Sollposition.vDone;

  o_TNC__stData.vDone;
  o_TNC__stRef.vDone;
  o_TNC__stRefCmp.vDone;
  o_TNC__bActiveCnt.vDone;
end;

//***********************************************************************/
// Initialisation
//***********************************************************************/
begin
  // Get and check SW-ID
  if (oID.csGetSwNo = "01 50") then
    csSymFileName := "../bin/sym/Toyota.sym";
  // elseif (oID.csGetSwNo = ".....") then
  // Add further symbol files
  // else
  else                                        // Happens also if Com-Port
                                              // isn't available
    Writeln("No valid sw-number");
    return;
  endif;

  // Open symbol file
  if !TOpen(fiSymFile,csSymFileName,nenFMRead) then  // Open sym file
    Writeln(GetErrorMsg(GetError));
    return;
  endif;

  // Init of symbol list
  if oSymList.poInit = nil then
    Writeln(GetErrorMsg(GetError));
    return;
  endif;

  // Set paramter for read symbol
  oSymList.vSetRdPar({wFlagsPar:}BDSYM_nBFReplDup|BDSYM_nBFIgnSynErr|
                                 BDSYM_nBFIgnExprErr,
                     {pstBitAddrDef:} nil,
                     {dwInvAddrPar:} 0);


  // Read Sym.File into Sym. List
  if(oSymList.enRead(fiSymFile) <> nenELSuccess) then
    oSymList.vDone;
    TClose(fiSymFile);
    Writeln("Read sym. File failed.");
    return;
  endif;

  // Init. of var - objects (for details see description of ADX-Library)
  // Parameters:
  // @oMemRdWrAccess:
  //            Read/Write object from ASSP3_IF
  //
  // oSymList.dwGetAddr("TNI_wFuelVolume",true):
  //            Reads the address from TNI_wFuelVolume, from the Sym. List
  //
  // TNI_wFuelVolume
  //           Symb. variable declared at this file

o_TNI_wFuelVolume.poInit(@oMemRdWrAccess,oSymList.dwGetAddr(
                         "TNI_wFuelVolume",true),TNI_wFuelVolume);
o_TNI__bStat.poInit(@oMemRdWrAccess,oSymList.dwGetAddr("TNI__bStat",true),
                         TNI__bStat);
o_TNI__bError_Cnt.poInit(@oMemRdWrAccess,oSymList.dwGetAddr(
                         "TNI__bError_Cnt",true),TNI__bError_Cnt);
o_TNI_biErr.poInit(@oMemRdWrAccess,oSymList.dwGetAddr("TNI_biErr",true),
                         TNI_biErr);
o_TND_wFuelVolume.poInit(@oMemRdWrAccess,oSymList.dwGetAddr(
                         "TND_wFuelVolume",true),TND_wFuelVolume);
o_TND__stData.poInit(@oMemRdWrAccess,oSymList.dwGetAddr("TND__stData",true),
                         TND__stData);
o_SMA__Sollposition.poInit(@oMemRdWrAccess,oSymList.dwGetAddr(
                         "SMA__Sollposition",true),SMA__Sollposition);
o_TNC__stData.poInit(@oMemRdWrAccess,oSymList.dwGetAddr("TNC__stData",true),
                         TNC__stData);
o_TNC__stRef.poInit(@oMemRdWrAccess,oSymList.dwGetAddr("TNC__stRef",true),
                         TNC__stRef);
o_TNC__stRefCmp.poInit(@oMemRdWrAccess,oSymList.dwGetAddr("TNC__stRefCmp",
                         true),TNC__stRefCmp);
o_TNC__bActiveCnt.poInit(@oMemRdWrAccess,oSymList.dwGetAddr(
                         "TNC__bActiveCnt",true),TNC__bActiveCnt);
end.
```