Stack Binding

Concepts

Revision: see Change Index

©

Büro für Datentechnik GmbH
D-35418 Buseck
Germany

Contents

0 CHANGE INDEX 3

1 ACRONYMS 4

2 INTRODUCTION 5

2.1 Diagnosis with BSKD 5
2.2 The AIDA System 6

3 BASIC FUNCTIONALITY (NORM AND ADX) 8

3.1 AIDA Stacks 8
3.2 The Object Oriented Approach 9
3.3 The Norming Object: NORM_toFormat 10
3.4 The Stack Binding 11
3.5 The Communication Object: ADX_toCommu 11
3.6 The Transfer Object: ADX_toTransfer 12
3.7 The General Access Object: ADX_toAccess 14
3.8 The Read/Write object: ADX_toRdWr 15
3.9 The Show Message Object: ADX_toShowMsg 16
3.10 The Message Objects: ADX_toMsg... 16
3.10.1 The Message Object, Type 0: ADX_toMsg0 17
3.10.2 The Message Object, Type 1: ADX_toMsg1 17
3.10.3 The Message Object, Type 2: ADX_toMsg2 18

4 THE IMPLEMENTATION OF DIAGNOSTIC FUNCTIONS 19

4.1 Example: Diagnostic Functions (Memory Read/Write, Free Telegram) 19
4.2 Adaptation of Message Objects (Read Identifier) 21

5 SYMBOLIC VARIABLES 24

5.1 Data Representation in the Diagnostic Application 24
5.2 The Variable Attendant Object: ADX_toVar 25
5.3 Implementation of Symbol Variables 25
5.4 Structured Symbol Variables 26

6 INDEX 29

0 Change Index

Date	Author	Rev.	Ref.	Type	Description
2007-03-30	Uwe Kühn	1.02.01	-	lang.	First English version

1 Acronyms

ADX
AIDA Data Exchange library.
A POOL module delivered with the AIDA Runtime System (POOL standard modules).

AIDA
Automotive and Industrial Diagnostic Assistance.
System that is used to implement computer supported diagnosis of control modules and bus systems.

AIDA Stacks
AIDA stack components library
A set of interface stack components delivered with the AIDA Runtime System.

BSK
BSK Datentechnik GmbH
The manufacturer of the AIDA-System.

BSKD
BSK Diagnosis
Generic automotive diagnostic program. The predecessor of AIDA.

ECU
Electrical Control Unit. Sometimes synonymous also named as controller or device.

NORM
A controller dependent standardisation library providing an object to translate data between host and ECU format.
A POOL module delivered with the AIDA Runtime System (POOL standard modules).

POOL
Portable Object Oriented Language. Object oriented programming language by BSK.
POOL is the programming language of the AIDA system.

RDM
Exemplary ECU used by BSK

2 Introduction

While to a large extent a monolithic binding of the addressable controller interfaces was realized in earlier diagnostic systems, the AIDA system builds on combinable interface components with their binding to the universal POOL Runtime. Like always, if a system is exposed to extended requirements, such a solution is appropriately more complex, but in addition, more consistent and more obvious in its impact. Since the AIDA-System replaces earlier diagnosis applications (BSKD/DOS, BSKD/NT, BSKD/2000), the changes are to be measured at these. The requirement, to describe the AIDA interface concept and also build a bridge to the earlier diagnostic programs, this document is to fulfil.

2.1 Diagnosis with BSKD

Just to remember: Former diagnostic applications made simple basic instructions available for data exchange with controllers, to which appropriate fixed telegram superstructures were assigned. In the beginning even these basic instructions were fixed and intrinsic application functions. Hereby such instructions were implemented as command contractions with subsequent parameters. The usually needed instructions here were e.g.:

ID
Read Identifier, read only, no parameters

MR addr,,len
Memory Read from given address and given number of data bytes

MW addr,data…
Memory Write to given address and consecutive data list

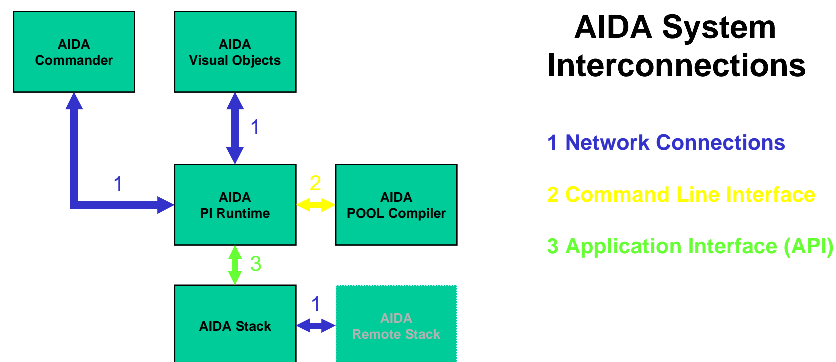
SR ident
Status Read with given identifier for desired status function

SW ident,data…
Status Write with given status function and consecutive data list

Advantages were the downright simple and short input of an instruction in a command line, disadvantages however the possible processing only byte by byte of the telegrams within macros and - which weighs most heavily - the only rudimentary existing support of different electronic devices or different application tasks.

2.2 The AIDA System

In particular the last requirement was not further to be fulfilled with the conventional concept. Therefore the object oriented abilities of the POOL-System were consistently used, in order to provide comparative functionality, without giving up the advantage of the simple command line input completely. In order to come to the point: The new approach with its various possibilities requires a fundamental re-orientation with the binding of controllers and goes far beyond the earlier concept.



From the illustration can be seen, how the individual functional blocks of the AIDA system are connected to each other. Hereby the POOL Runtime and the tied up driver stack still represent the closest connection, while the AIDA Commander and the graphics extensions are connected via general network interfaces.

The substantial separation between stacks and POOL application is also strict, where protocols for the data communication are treated exclusively within the stacks, while actual telegram contents are matter of the applications and/or appropriate adaptation modules. This document describes these adaptation modules.

Anyhow, a prior goal in the AIDA concept is not to deal with actual data exchange functionality, but more to understand data of arbitrary kind as objects and then to provide them with the appropriate data access methods. So, for example, it should be all the same for the user whether the needed data lies in the RAM, ROM or NV-RAM of the electronic device. The user doesn’t wants to specify the transmission path with each access, but simply access data contents.

Example:

```
oID.boRd
```

would be a correspondence to the former ID instruction of earlier diagnostic programs, whereby in the new paradigm it acts as an ID objects, over it's read method the data contents to be provided. This object has also one presentation method, consequently called like

```
oID.vShow
```

Since such an ID object may belong to an individual device and probably some other controllers also possess an ID object, the module name which the respective ID object is part of can be placed for clarification and distinction in front of the object identifier. Reasonably the appropriate POOL module here would get the name of the device, because here the objects relevant for this device are defined. For the exemplary case that this equipment would be called "RDM", the appropriate adjustment module then became "RDM.pool" and the ID object stated in the example could then purposefully be addressed by

```
RDM.oID.boRd; RDM.oID.vShow
```

What conceptionally applies to special diagnostic functions, is naturally also to be applied to symbolic variables in the controller, which consistently were designed with a corresponding objects, so that the writing access to such a variable then may act like

```
wKFactor := 5448; o_wKFactor.boWr;
```

if one assumes that the variable in the device was created under the name "wKFactor", being of the type 16 bit unsigned integer (Word) and has to be adjusted to the value of 5448. It's obvious that the user not any longer must know whether the access itself is now to the RAM or to the NV-RAM of the ECU's memory. The access rules are implemented once in the corresponding data object and then only the symbolic address will be updated by a suitable symbol generator of the development tools.

In the following the basic objects necessary for this purpose, provided in the supplied modules of the stack binding, become describes.

3 Basic Functionality (NORM and ADX)

Following the stepwise binding of interfaces becomes described at the example of diagnostic functions with the means of the POOL Runtime system. Naturally neither any of the interface components nor the POOL system itself is limited to diagnostic functionalities, these were chosen, however, as an example of a general application and for the comparison with earlier concepts.

The objects represented here are all together provided by the ADX and the NORM-module.

3.1 AIDA Stacks

In the AIDA system the pure interface specific parameters are completely sourced out to the AIDA driver stacks. Although the complete set up of stacks as well as the tuning of their parameters are manageable out of a POOL application, there is, however, the substantial simpler way of doing with the supplying program "AIDA Stacker", that helps for stacking, for setting of parameters and then store them all as a stack configuration. This configuration in total then becomes loaded by the POOL application, to which no more than a single command line is necessary in substantial. One further advantage of this procedure is, that a such way created stack configurations can be easily used again in related applications. For this it is surely meaningful - and urgently recommend - to store these configuration files in a more central place and to make all workspace dependent specific setups, e.g. the used COM port, accessible by the use of environment variables.

Thus here the stack concept is not further described; it is only important to know in this context that the elementary parts of communication, i.e. lower and upper protocol layers, are treated within the stack and only the pure application data reaches the application layer. Thus the actual structure of telegrams and what still belongs to it, like timing or higher mechanisms, are unimportant for the application and even completely unknown to it. On the other side, however, special functions, like ID mentioned above, or MR, MW etc., are not realised in the stack functionality and must be implemented as part of the application.

This is undisputed the correct deployment of that functionalities, where with the stacks also non-diagnostic extents are to be treated by the AIDA system.

The following line shows an example to the elucidation of an abstract, but quite usual structure of telegrams:

Transmitter	Receiver	Telegram length	Function Code	Data ...	Checksum

In this structure of a data telegram those grey back grounded components become administered by the stack components themselves, while the application only contributes the pure utility data, in this case the function code and further data. It is apparent that the actual diagnostic functions come to lie in that brightly marked fields, thus must be treated within the application interface.

This is, however, at the same time already that level, on that former diagnostic programs developed their commands, as they coded the actual instruction into the function code and inserted the data (with MR this e.g. would be address and length). First this naturally is possible also in POOL and nobody can and should be (or, as one will see, better nevertheless) prevented from doing this way and to build himself a procedure named MR that implements accurately, what the earlier instruction made – however, now nevertheless in POOL syntax, thus about:

```
MR (0xFF48,10,'nh')
```

or, if one likes, also symbolically

```
MR (kFactor, 2)
```

With such low demands hereby the problem would be settled and this document thereby already concerned at the end. In reality, naturally, thereby nothing is won and one would have saved the efforts for AIDA at all.

3.2 The Object Oriented Approach

The crucial lack of the aforementioned implementation above all consists of the fact that to the storage location mentioned first no further contentwise linkage exists. In that symbolic variant nevertheless still the address into the storage area of the controller is known, but neither the controller specific arrangement of data in the memory or the size of the date "kFactor" is well-known to the POOL application. In that mentioned example the read operation at the address of "kFactor" only results in an accumulation of 2 bytes given back, while in the controller the data possibly may be represented as a genuine Word type (16 bit unsigned int) and is subject to a certain byte order (e.g. “little endian” = low byte first or "big endian " = high byte first) for which, however, the processor of the device itself does not care in the slightest, because its architecture this simply determines. For POOL applications this is, however, very probably of importance, because the processor’s architecture of the underlying host system will possibly follow completely a different byte order.

If we are about to do so much effort, then at least one of the goals must be, the date "kFactor" to be exactly used in the diagnostic application like the device also does, i.e. the symbol "kFactor" must be at the disposal as a variable of the same name with a suitable data type and there must exist some methods, which make it possible to adjust their appropriate contents with that of the device.

With the term of the "methods" one is purely linguistically already in the realm of object oriented programming and therefore it seems likely, for each variable to place aside a suitable object with also appropriate methods, which takes over those demanded tasks. In every case such an object avails itself of such methods, that memory in the device can be read or written, eventually. That this succeeds, however, some steps are necessary in advance.

3.3 The Norming Object: NORM_toFormat

First it is to be ensured that data structures, which are about to be transferred to unstructured storage areas, to be put down in the correct byte order, thus the respective processors can access to directly. For this a special standardisation module named "NORM" exists in the POOL libraries, which contains the necessary object "NORM_toFormat" and makes this available for application purposes.

```
type
  NORM_tenByteOrder = ( NORM_nenBOUnknown, NORM_nenBOBigEndian, NORM_nenBOLittleEndian );
  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: tpstTypeRec;
                             pvData: Pointer); virtual;
    procedure    vTranslateBType (enDirection: NORM_tenDirection; stUAVar: tstUArg); virtual;
    procedure    vCollectData (var bsData: ByteString; ..);
  end;
  NORM_tpoFormat = ^NORM_toFormat;
```

An application must create only one instance of this object for similar controllers and initialize it with the desired byte order. The translate methods in these objects are not relevant for the application level, but will be used exclusively from separate objects, that serve themselves of this format object. The creation of an instance of this object is very simple, example:

```
var
  oFormat: NORM_toFormat;  { ECU data format object }
...
begin
  oFormat.poInit (NORM_nenBOLittleEndian);   { initialisation to little endian byte order }
...
```

On the other hand the CollectData method is (possibly also for application purposes) expressed important, to return arbitrary in the handed over list specified parameters with consideration to convert the byte order of the controller into a data stream back to bsData.

For adjustments, which go beyond pure byte conversions (e.g. different number of bits in mantissa and exponent with real numbers) the object-oriented model supplies

likewise equal the suitable approach. For this it is only to derive an own type of object and to overwrite the conversion method by one own method, which exactly treats such cases. Everything else, in particular the application, remains unaffected from that.

3.4 The Stack Binding

As previously mentioned, it is necessary that the application can avail itself of a pre-configured stack. For this the associated configuration file becomes loaded by means of the AIDA_hRestoreStack-Funktion from the AIDA module and the final stack becomes assigned to a handle (here in the example named "hStack").

```
var
  hStack:        tHandle;        { ECU stack handle }
  csStackIdent:  CharString;     { AIDA stack identifier }
  dwSLevelMask:  DWord;
...
procedure vInitStack (var hStack: tHandle; csFileName: CharString);
begin
  hStack := AIDA_hRestoreStack (csFileName, csStackIdent, dwSLevelMask);
  if hStack = nil then
    Writeln ('Error Loading Stack');
    Halt (nExitAppErr1);
  endif;
end; { vInitStack }
```

Thus from the AIDA module the available functions are accessible via the stack handle and in principle data exchange with the controller can take place already. Since the AIDA system however has been constructed as an event driven multi threading system, this binding is not completely as trivial and this results in the necessity for a further fundamental object:

3.5 The Communication Object: ADX_toCommu

The communication object as well as further auxiliary objects is to be found in the module ADX (**A**IDA **D**ata E**x**change). The object first provides a very simple method, that permits to send and receive arbitrary data blocks over the stack. This method named "boGenericMessage" expects only a scratchpad memory for the sent data and supplies the answered data in a likewise handed over buffer again. Hereby the function makes use of two ByteStrings of dynamic size which are best suited for such a task. Two further methods for Connect and Disconnet as well as an ErrorHandler supplement this object. The FreeMsg method makes it possible, to dispatch rudimentary telegrams already on this level.

```
ADX_toCommu = object
  bFlags:     Byte;           { flag byte }
  dwFlags:    DWord;          { AIDA stack flag word }
  hStack:     tHandle;        { stack handle of desired commu channel }
  poFormat:   NORM_tpoFormat; { pointer to associated byte order object }
  bsData:     ByteString;     { local data buffer }
  bRetry:     Byte;           { no. of retries for re-connections }
 private
  bRetryCtr:  Byte;           { local retry counter up to bRetry }
 public
  constructor poInit (hStack_l: tHandle; poFormat_l: NORM_tpoFormat);
  function    boConnect: Boolean; virtual;
  function    boDisconnect: Boolean; virtual;
  function    boErrorHandler (dwError: DWord; pstEvent: AIDA_tpstEvent): Boolean; virtual;
  function    boGenericMessage (var bsTrData: ByteString; var bsRcData: ByteString): Boolean;
  function    boFreeMsg (..): Boolean;
end;
ADX_tpoCommu = ^ADX_toCommu;
```

In principle the communication object also is not relevant for applications, but provides only basic functionality for objects, which implement the actual access functions. For further use an additional basic object becomes needed, that makes possible to provide the data of boGenericMessage also with an internal structure.

3.6 The Transfer Object: ADX_toTransfer

This object essentially provides the public parameters, which make it possible to place information such as addresses, lengths and function codes to the desired positions within the data block. The likewise contained method "Transfer" on the other hand is consciously defined not public and so available only object-internally. As already said, the object assumes the generalized case, where the utilizable data of a telegram transmission consists of a heading with possible information such as function code, address and length and then in further of unstructured data. Hereby the transfer object is constructed in a way that, depending upon necessity, those mentioned header data may be contained all or not or possibly partly only.

```
const  { ADX_toTransfer.bFlags }
  ADX_nTFSegmentedRead    = 0x01;  { segmented read access to ECU data allowed  }
  ADX_nTFSegmentedWrite   = 0x02;  { segmented write access to ECU data allowed }


ADX_toTransfer = object
    bFlags:     Byte;             { flag byte                                  }
    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  }
    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_tpoTransfer = ^ADX_toTransfer;
```

The local parameters of the object contain a flag byte for transfer options as well as two parameter sets, one for a transmit telegram and one for a receive telegram. With the latters in each case …Cmd describe the function code, …DataPos the position of the first of the unstructured data and …Order the insertion sequence of function code, addressing and length information. The following example of a data read order makes clear, how to use the parameters:

```
oReadMemX: ADX_toTransfer = (
  bFlags : 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))
);
```

First it is indicated in the flag byte that segmented read access is permitted. That means that read inquiries, that are longer than what the controller can deal with in one transferred telegram may be divided into multiple telegrams. 'dwTrCmd' contains the function code of the read order. 'bTrDataPos' indicates the position of the first data byte, that does not belong to the header data. Finally the positions of the data bytes are described in 'unTrOrder', to which the header information belongs. As seen from the type definitions evidently, all header information is implemented as DWord data types, thus maximally consisting of 4 bytes. In accordance 'unTrOrder' describes, what positions the individual bytes take in the header information of the handed over buffer to the stack. The example shows that the lowest ordered byte of the Function code (Command) comes to lie at position 0 of the transmit buffer. The other bytes in each case contain positions of 0xFF, which would come to lie behind 'bTrDataPos' and thereby will find no consideration. The next assigned value is the length specification (DataSize), their lowest byte is set at position 1. The address information, eventually, comes to lie at positions 2 and 3, hereby the higher ordered byte first.

For the answer telegram the parameters show that the same command is expected as function code and that besides the function code, which is to be found at position 0 of the receive buffer, the requested data to be found starting from position 1.

That the object is apparently oversized for the use of free telegrams, but evenly also may consider such examples, shows the following case. In this case all header information are actually irrelevant, because they are not to be inserted at all. Yet also such kind of communication nevertheless belongs to this level of treatment.

```
oFreeX: ADX_toTransfer = (
  bFlags : 0;
  dwTrCmd: 0;
  bTrDataPos: 0;
  unTrOrder: (
    unCommand:    (abA: (0xFF,0xFF,0xFF,0xFF));
    unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
    unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
  );
  dwRcCmd: 0;
  bRcDataPos: 0;
  unRcOrder: (
    unCommand:    (abA: (0xFF,0xFF,0xFF,0xFF));
    unDataSource: (abA: (0xFF,0xFF,0xFF,0xFF));
    unDataSize:   (abA: (0xFF,0xFF,0xFF,0xFF))
  )
);
```

The example thereby shows for a free telegram also such one case, with which it is not clearly evident whether by the execution of the transmissions data contents to be given to the controller or fetched from it. With other, more dedicated telegrams this is mostly clear. Thus e.g. the requirement of the controller identification (ReadID) would only expect data answered, but certainly none send. More typical actuator tests (SetStatus), however, usually transmit only data, but expect none back. The transfer object is therefore designed in a way that it either does not need the knowledge over that.

Further considerations show that it is appropriate, to divide diagnostic functions into a class, their purpose it is to implement arbitrary individual commands for which there are no complementary functions (like ReadID, SetStatus, GetStatus and others), and a class with complementary functionality in each case (Read/WriteMem, Read/WriteFlash among others). The latter will always have the goal to access the data by addressed reading and writing. The two classes mentioned become represented by two further objects: One general access object and one read/write object, which both merge the transfer object themselves and then make use of it’s transfer method.

3.7 The General Access Object: ADX_toAccess

This object first contains a pointer to the corresponding communication object and integrates, as announced, its individual transfer object. As the only method stands boXchgData for the order, which permits to hand over an extended identifier and a length information as additional header data, while it otherwise handles only one scratch buffer for the transmit data and the likewise also handed over receive buffer with the receive data filled up.

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

The object is not especially designed for write or read operations, but can fulfil both purposes depending upon use. With such an object yet one is already within easy reach of diagnostic functionalities such as ReadID and GetStatus, SetStatus etc.

3.8 The Read/Write object: ADX_toRdWr

As previously mentioned, also write and read operations may be accomplished with the aforementioned object. For genuine memory read and write, however, it is essential for an access object to know that it deals with complementary operations on the same data area. Otherwise (possibly necessary) read-modify-write operations could not be performed, unless it’s clear which writing procedure corresponds directly with a reading access. In addition it happens that sometimes controllers are subject to strong restrictions, which concerns the access to their own address range.

```
ADX_toRdWr = object
  poCommu:    ADX_tpoCommu;    { pointer to associated communication object }
  dwAddrOffs: DWord;           { global address offset for read/write ops   }
  bAddrShift: Byte;            { address shift for read/write operations    }
  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         }
  dwWrBlkMsk: DWord;           { mask for legal write address ranges        }
  oReadX:     ADX_toTransfer;  { data read object                           }
  oWriteX:    ADX_toTransfer;  { data write object                          }
  constructor poInit (poCommu_l: ADX_tpoCommu);
  function    boRdData (var bsRcData: ByteString; dwSource: DWord; xoLen: tSize): Boolean;
  function    boWrData (dwDest: DWord; bsTrData: ByteString): Boolean;
end;
ADX_tpoRdWr = ^ADX_toRdWr;
```

Since it is here, as described, about general addressed data accesses, the object contains in opposition to the simpler access object a set of additional parameters as well as in each case two transfer objects, i.e. one for reading by the dedicated boRdData method and one for the dedicated writing by means of boWrData. This object is thus a fundamental pre-condition for the at the beginning aimed goal of the symbolic access to controller data.

First, however, the so far designed objects shall serve to implement the earlier used diagnostic functionalities in the now object-oriented manner. For this purpose general message objects exist, which serve themselves of the one or the other class of access objects, respectively.

3.9 The Show Message Object: ADX_toShowMsg

Since tasks such as memory write and read are not self purpose, but usually come with the sifting of that transferred data, it seems likely to pre-define an own show object for the representation of such data. The reason is obvious: In particular the typical hexadecimal output of data in connection with memory accesses will not differ for the cases, in which either RAM, ROM, EEPROM, Flash or still completely different data areas are accessed by means of different access objects. Therefore a uniquely useful defined object for all such output can be used.

```
ADX_toShowMsg = object
  wFlags:          Word;         { flag word, see ADX_nSF*           }
  bHdrLimit:       Byte;         { output header                     }
  i8IdentDigits:   Int8;         { display ident as title            }
  cIdentFormat:    Char;         { format for dwIdent                }
  bWidth:          Byte;         { width for one byte                }
  bCnt:            Byte;         { no. of bytes per line             }
  bDummy:          Byte;         { reserve                           }
  csTitle:         String;       { title for data output             }
  procedure vShow (dwIdent: DWord; var bsData: ByteString);
end; { ADX_toShowMsg }
ADX_tpoShowMsg = ^ADX_toShowMsg;
```

With the different possibilities of format adjustment, as they can be given in the parameters of this object, is not to be dealt here. The explanations concerning the parameters are to be found in ADX.PLI accordingly. Enclosed only one example, how a meaningful data output for memory accesses can be parametrized:

```
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:         ''
  );
```

3.10 The Message Objects: ADX_toMsg...

The message objects finally needed, in order to be able to implement special diagnostic functionalities, are all derived from a basic object, that provides data buffers for sending and receiving and additionally performs a general show method for the presentation of the received data.

```
ADX_toMsg = object
  bsTrData:   ByteString;            { transmit buffer                         }
  bsRcData:   ByteString;            { receive buffer                          }
  stError:    ADX_tstError;          { error information                       }
  poShow:     ADX_tpoShowMsg;        { corresponding show message object       }
  constructor poInit (poShowPar: ADX_tpoShowMsg);
  procedure   vShow (..); virtual;
end; { ADX_toMsg }
ADX_tpoMsg = ^ADX_toMsg;
```

Derived from this and including the prepared access and write/read objects one receives an own message objects, with which the far most diagnostic functions are realizable. They all use methods named boRd and boWr, which then are meant for data read and/or write accordingly.

3.10.1 The Message Object, Type 0: ADX_toMsg0

This object neither in the read method nor in the write method expects any dedicated parameters (therefore its type 0), i.e. that no further parameters are needed and only the function code is used by the referenced ADX_tpoAccess object.

```
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;
```

Both methods, however, can work on an open list, so that arbitrary data can be transferred here. The user deliberately decides whether he rather wants to use the read or the write method depending upon application.

Typical application for such an object is the functionality of ReadID (whereby here the technically just as possible write method would rather not be used from obvious reasons).

3.10.2 The Message Object, Type 1: ADX_toMsg1

The object of type 1 expects one additional fix parameter in the methods boRd and boWr and is otherwise derived from the type 0 object.

```
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;
```

Such objects are typically used by status functions, whereby the function code of the linked ADX_tpoAccess object implements the diagnostic functionality of GetStatus or SetStatus, while the handed over "dwIdent" describes the subfunction code of the status function. Since it usually concerns of non-complementary functions, only one of the provided read or write methods available will be probably used, respectively.

3.10.3 The Message Object, Type 2: ADX_toMsg2

This object needs *two* parameters in its read method, which are explicitly implemented as address and length. Thus in particular its use shows up in connection with memory accesses. boRd and boWr here really are complementary accesses and accordingly this object also uses the read/write object introduced above for data exchange.

```
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 mentioned last is stored in 'dwLastAddr', among other things for the purpose to achieve a reasonable data output within the show method.

4 The Implementation of Diagnostic Functions

With up to here provided objects the substantial functionality of the earlier diagnostic programs would be reached. We summarize:

Objects can be defined, which implement the necessary diagnostic functions, however with some advantages:

- No fundamental diagnostic functions are pre-defined anymore, on the other hand arbitrary new functions can be implemented easily in addition. In particular name choice and kind of execution are dedicated to the developer.
- Each diagnostic function has its own buffers and also its own show method. The latter is particularly important with data interpretations like the evaluation of e.g. ReadID data (which highly differ between different devices).
- Each diagnostic function is assigned to its own device, so that in advance to an access no more interface parameters must be changed. This advantage is simply provided by concept.
- The object-oriented approach permits to overwrite inherited access methods or give new in addition in a simple manner.
- The consideration of the byte order of the controller is already implemented on the lowest level.
- The necessary tunings can be made easily and clearly in the source text of the binding module.

4.1 Example: Diagnostic Functions (Memory Read/Write, Free Telegram)

The following example shows a device binding, which implements some substantial diagnostic functions. The complete example is to be found in similar way in the module "rdm.pool". In principle it might be meaningful, as in the example happens, to page out the device binding in an own module with the name of the device.

The individual implementation steps always obey the following pattern:

1. Declaration and initialization of a normalisation object;
2. Declaration of a stack handle and loading of the stack configuration;
3. Declaration and initialization of the communication object;
4. Declaration and initialization of the message objects.

The whole project requires only few lines of code. First when merging the necessary modules,

```
import AIDA;
import NORM;
import ADX;
```

then the creation of the objects and the interface handle in the declaration part

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

  oFree:        ADX_toMsg0;     { object for free messages (type 0)         }
  oSWr:         ADX_toMsg1;     { object for status write operations (type 1)}
  oMem:         ADX_toMsg2;     { object for memory access (type 2)         }
```

and finally their initialization in the instruction part

```
begin
  oFormat.poInit (NORM_nenBOLittleEndian); { Step 1:  data access format for ECU }
  vInitStack (hStack, 'MyECU-Stack');      { Step 2a: loading driver stack }
  oStack.poInit (hStack);                  { Step 2b: initialising stack object }
  oCommu.poInit (hStack, @oFormat);        { Step 3:  initialising communication object }

  oFree.poInit (@oFreeMsg, @oCommu, @oFreeShow); { Step 4a: initialising free message object }
  oMem.poInit (@oRWMBA, @oCommu, @oMemEEPShow);  { Step 4b: initialising memory access object }
  oSWr.poInit (@oSetStatus, @oCommu, @oSWrShow); { Step 4c: initialising status write object }
end.
```

Hereafter the most substantial functions for the execution of diagnostic commands are available. Examples:

Write Memory:

```
oMem.boWr (0xFFB8,0x33)                       { writing one byte }
oMem.boWr (0xF130,1,2,3,4,5,6,7)              { writing a couple of bytes }
oMem.boWr (0x1200,0x1234,0x5678)              { writing two words }
oMem.boWr (wADC0Address,Word(0x80))           { writing one word at symbolic address location }
oMem.boWr (wKFactorAddress,wKFactor)          { writing a symbolic value }
oMem.boWr (wMotorFieldAddress,aabMotorField)  { writing a motor parameter field in total }
oMem.boWr (wOdoBlock,wKFactor,bDivider)       { writing a mixed parameter list }
```

Please note that the write method, that has been designed as a function can be executed actually also as a procedure, if one is not interested in the return value, which indicates success to the execution.

The actual power of the made implementation, however, lies in the fact that because of the run time type information, which is inherent to the POOL compilation, not only mixed lists of completely different parameters can be handed over, but even each individual of these parameters is still converted into the correct byte order of the device.

Memory Read:

```
oMem.boRd (0xFFB8,10)                         { reading one byte }
oMem.boRd (wOdoBlock,10)                      { reading ten bytes at symbolic address location }
```

Here the requested storage area is read in each case and put down into the associated receive buffer “oMem.bsRcData”. The appropriate data representation of the storage

area then possibly takes place via (the overwritable) show method, which causes a hexadecimal notation with this type of object, e.g.:

```
oMem.vShow
00001234:  11 98 7E 78 73 70 46 06 5E 00  "..~xspF.^."
```

Free Telegram:

```
oFree.boRd ('R',10,0x12,0x34)
```

Like in the preceding example the free telegram causes a memory read from the same address. However, an oFree object does not know anything about contents of the data received back (otherwise it would not be really free). Therefore the show method doesn't show the start address (0x1234) and the data also still contain the retrieved function code (0x52).

```
oFree.vShow
00000000:  52 11 98 7E 78 73 70 46 06 5E 00  "R..~xspF.^."
```

With the oFree object, as previously mentioned, it is no matter whether the read or the write method is used. Here just the user selects on the basis of the criterion whether rather data become sent (Write) or fetched (Read).

4.2 Adaptation of Message Objects (Read Identifier)

Apart from the relatively general cases of memory reading and writing and the dispatching of free telegrams, the status change functions are usually very system specific. A special object in any case is that one, which picks the device identification (in former times the ReadID function), because at least the show method cannot be generalized since the manufacturer's specific characteristics are defined in such a way that the received information cannot always given out in a directly readable way. In this case such an object would be derived from the prepared objects, but overwrites the show method. Again for that purpose only simple instructions are necessary. First the parameter set up, approximately in the kind:

```
{ read identifier parameter set }
  oRdID: ADX_toAccess = (
    poCommu: nil;
    oTransX: (
      bFlags : 0;
      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))
      )
    )
  );
```

Afterwards the derivation of the ID object with own show method,

```
type
{ type 0 message object with special show method }
  toID = object (ADX_toMsg0)
    procedure vShow (..); virtual;
  end;
```

the initialisation of a suitable show object,

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

finally the declaration of the object instance as a variable,

```
var
  oID: toID;      { object for ID access }
```

and the implementation of a suitable show method

```
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;
```

as well as its following initialization.

```
…
  oID.poInit (@oRdID, @oCommu, @oIDShow); { Step 4d: initialising read ID object }
…
```

This adaptation must take place maximally once per device class and is neither difficult nor too expensive. The use of the object then happens via the call

```
oID.boRd; oID.vShow
```

and will show the following output:

```
     SWNo  HWNo  DD   MM   YY
ID:   10    14   12   05   95
```

That the output of the device identifier (contrary to former times) has to be performed in two steps (Read, Show), is no real disadvantage, because finally the read information is further locally available and can be evaluated as desired for other purposes, too. If nevertheless (e.g. because of extensive use in the command line) only one single call is desired, there is naturally no hesitation to call also the read method from the adapted show method.

5 Symbolic Variables

With this section we return to the preview expressed under 2.2 on the symbolic use of controller variables.

After compiling and linking of the system software variables in controllers become values within symbol tables. Every entry here contains the symbolic name of the variable, its base address in the assigned storage area and, if necessary, further possibly even complex type information. Using the minimum information (name, value) in earlier diagnostic programs controller variables already could be symbolically addressed by memory read and/or write. However the disadvantage remained that the here addressed data acted only as a poor copy of the controller memory. I.e. the diagnostic application needed additional information about the type of the appropriate variable and its byte order in the controller. Beyond that the direct use of the data within the diagnostic program was not possible.

5.1 Data Representation in the Diagnostic Application

The first step thus would have to lead to the fact that in a diagnostic application the variable of a controller should be available under the same name on compatible data types. If the symbol tables (or cross-reference lists) of the controller contained those necessary information, it is first no larger problem by an external symbol generator to provide a .pool file, which defines this variables for POOL language accordingly. Hereby the symbol generator will make use of usually available basic types. Example of a conversion of the variables of the Module ODO (in C) of a device into POOL:

```
word  ODO_wKFactor          →        ODO_wKFactor:  Word;
byte  ODO_bFlags            →        ODO_bFlags:    Byte;
int16 ODO_i16Counter        →        ODO_i16Counter Int16;
```

POOL is designed in a way that appropriate compatible types are made available for in controllers usually existing basic data types. With that in a diagnostic application on the one hand variables could have the same symbolic name to be accessed as in the ECU and in addition the variable has the same range of values. Neither the controller nor the diagnostic program must take consideration of the kind of order in its own memory when using the variables.

That changes instantly, when data from the controller has to be read in order to adjust it with the local copy, and/or when storing data into the controller. Here it must be very probably considered during the transmission if the byte order between the diagnostic unit and controller is identical or different. In addition it is naturally also of importance, at which address the data in the controller comes to lie. Such information cannot be stored inside the variable, but must be held in other locations.

5.2 The Variable Attendant Object: ADX_toVar

For the description of the characteristics of controller variables therefore one special attendant object was developed, which contains this necessary information to manage the data exchange with the actual controller (ECU).

```
{ ECU dependent variable object }
{ This object gives a local representation of a single ECU data element.    }
{ The data image itself can be manipulated directly via access to the       }
{ identical named data structure. Whithin this object this data image will  }
{ be addressed by 'pabData'. The structure of the data image is described   }
{ by 'pstType'. It is obvious, that this object must use a Rd/Wr object     }
{ for ECU data access.                                                      }
ADX_toVar = object             { local object type for ECU variables       }
  pabData:    tpaByte;         { pointer to local data image               }
  pstType:    tpstTypeRec;     { pointer to local data type structure      }
  poRdWr:     ADX_tpoRdWr;     { pointer to data rd/wr object for this var }
  dwSource:   DWord;           { data source address within ECU            }
  { initialisation }
  constructor poInit (poRdWr_l: 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    }
end;
ADX_tpoVar = ^ADX_toVar;
```

The object first contains a pointer to the local copy of the controller variable. This pointer is for reasons of simplification implemented as a pointer to a byte array. One further pointer points to the associated data type description block. The third pointer addresses the associated read/write object from chapter 2, over which the actual reading and writing again of data contents takes place. As last information the object contains the source address of data contents in the controller.

In addition there are two methods “boRd” and “boWr”, with which entire variables can be read from and/or be stored to the controller. In this way the data exchange between the diagnostic computers and the controller is provided as desired.

5.3 Implementation of Symbol Variables

The implementation of such symbol variables, if this happens by hand, is some boring clerical work and is justified only if it deals with only some few variables. Usually however this work might be fulfilled by a symbol generator. This one creates the variables in a separate module, which has to be tied to the diagnostic application. In the so far selected example such a file e.g. became named like “MyECU_Sym.pool”.

For each individual variable three steps are necessary in this module:

1. Declaration of the variables;
2. Declaration of the associated variable attendant objects;

3. Initialization (and de-initialization) of the variable attendant objects.

```
var                           { Step 1: declaration of the ECU variables }
  ODO_wKFactor:  Word;
  ODO_bFlags:    Byte;
  ODO_i16Counter Int16;

var                           { Step 2: declaration of corresponding attendant objects }
  o_ODO_wKFactor:  ADX_toVar;
  o_ODO_bFlags:    ADX_toVar;
  o_ODO_i16Counter ADX_toVar;

…

{ vDeinit: Module deinit }
procedure vDeinit;            { Step 3b: de-initialisation of attendant objects }
begin
  o_ODO_wKFactor.vDone;
  o_ODO_bFlags.vDone;
  o_ODO_i16Counter.vDone;
end; { Deinit }


{ Module init }               { Step 3a: initialisation of attendant objects }
begin
  o_ODO_wKFactor.poInit   (@MyECU.oRWMBA, 0x1234, ODO_wKFactor);
  o_ODO_bFlags.poInit     (@MyECU.oRWMBA, 0x2345, ODO_bFlags);
  o_ODO_i16Counter.poInit (@MyECU.oRWMBA, 0xFFB8, ODO_i16Counter);
end. { Init }
```

One should consider the selected nomenclature for the designators of the attendant objects. While for POOL the observance of the demanded nomenclature in the Language Description document is urgently recommended, it cannot be presupposed that these or an approximately similar nomenclature will be used also within the controller. For the controller variables that first is uncritical, as long as no global trivial names become used there, which may collide with variables of the diagnostic application. Using the recommended nomenclature regulation, which prefixes the module names (like here in the example ODO) this risk is minimal. It is advisable, of course, to maintain even the same name for the associated variable attendant object as for the variable and only to prefix it with an “o_”. Thus on the one hand it is achieved to be seen already by the name, which data structures belong together and secondly the attendant object becomes clearly characterized in accordance with the POOL nomenclature. By confinement to a simple nonspecific “o_”, unnecessary clerical work becomes avoided, whereby the underline already optically obtains a differentiation to the original name.

5.4 Structured Symbol Variables

The use of structured symbol variables follows completely and analogous to the unstructured ones. The only exception is that the symbol generator (or the user) has to create and to name a suitable equivalent structure of the controller variables in advance. The other steps, like declaring and initializing that variable and the associated attendant object as well, are alike.

```
type
  ODO_tstTripStruct = record                 { Step 0: structured equivalent type definition }
    bFlags:   Byte;
    dwValue:  DWord;
    pstPtr:   ^ODO_tstVarBuffer;
    adwValue: array[0..3] of DWord;
    enEnum:   tenNPRes;
  end;

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

var
  ODO_stTripStruct:    ODO_tstTripStruct;     { Step 1 }
  ODO_stVarBuffer:     ODO_tstVarBuffer;

  o_ODO_stTripStruct:  ADX_toVar;             { Step 2 }
  o_ODO_stVarBuffer:   ADX_toVar;

…

{ vDeinit: Module deinit }                   { Step 3b }
procedure vDeinit;
begin
  o_ODO_stTripStruct.vDone;
  o_ODO_stVarBuffer.vDone;
end; { Deinit }


{ Module init }                              { Step 3a }
begin
  o_ODO_stTripStruct.poInit (@RDM.oRWMBA, 0x1234, ODO_stTripStruct);
  o_ODO_stVarBuffer.poInit  (@RDM.oRWMBA, 0x1234, ODO_stVarBuffer);
end. { Init }
```

Contrary to the unstructured symbol variables here the additional task is set to work on individual fields of the structure, occasionally. This is considered by the attendant object in reality providing a set of further methods for this purpose, as well.

First there are the methods boRdField and boWrField, that permit individual sub-structures, addressable by field names, to be read and written. In each case the desired individual parts from the associated overall structure becomes handed over as a parameter.

```
  { read/write functions for partial read/writes of the data structure }
    function    boRdField (var xParam): Boolean;
    function    boWrField (var xParam): Boolean;
```

These methods assume that partial accesses to the data is actually permitted. One of the settings in the flag byte of the read/write object indicates that.

Beyond that there exist a couple of methods (...Get and vPut...), which provide arranged access even then if there is no unambiguous type conversion for the created symbol variable. That e.g. happens for C unions, which are excluded from the automatic normalisation by boRd and boWr, because it cannot be assured that the normalisation is consistently feasible for all possible variants.

```
{ read access to local data buffers 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 buffers 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);
```

Here the mentioned Get and Put methods have exclusive meaning for local access and do not lead explicitly to a reading or writing access to the controller, for what however the aforementioned methods come into question.

6 Index

A

ADX 8

B

Basic Functionality 8

C

Change Index 3
Contents 2

D

Diagnostic Functions 19

I

Index 29
Introduction 5

N

NORM 8

S

Symbolic Variables 24