Automotive and Industrial Diagnostic Assistance

Concepts and Overview

Revision: see changes index

©

Büro für Datentechnik GmbH
D-35418 Buseck
Germany

Contents

0 MODIFICATION HISTORY ... 3

1 MOTIVATION ... 4

2 STRUCTURE OF THE AIDA SYSTEM ... 6

3 POOL ... 8

4 AIDA DRIVER-STACKS ... 10

4.1 Basics ... 10
4.2 The AIDA Interface Driver Concept ... 11
4.3 The Structure of AIDA-Drivers ... 16
4.4 The API of the AIDA Interface Drivers ... 17

5 AIDA VISUAL OBJECTS ... 19

6 AIDA COMMANDER ... 22

6.1 Interactive Configuration ... 22
6.2 Application Controlled Configuration ... 26
6.2.1 Instructions for the Creation and Parameterization of Windows ... 26
6.2.2 Instructions for the Set-up and Execution of Menus and Submenus ... 26
6.2.3 Instructions for the Creation of Dialog Boxes with Related Controls. ... 26
6.2.4 Instructions for the Creation of Toolbars ... 26
6.3 Comfort Functions ... 27
6.4 Development Devices ... 27
6.5 Help System ... 28
6.6 Debugging ... 28
6.7 Data Objects ... 29
6.8 Data Types ... 30
6.9 Procedures (Macros) ... 31
6.10 POOL-Threads ... 32

7 INDEX ... 33

0 Modification History

Date	Author	Rev.	Ref.	Type	Description
2008-09-18	Uwe Kühn	2.03.00	all	content	Review and formatting
2007-02-23	K.-H. Damm	2.02.00	various	content	Updated examples: adapted incorrect POOL type prefixes and AIDA Component names.
				content	Revised wording and phrasing in various sections.
2007-02-22	Uwe Kühn	2.01.00	all	editorial	Translation to English language based on “AIDA Konzept.doc” as of 2001-06-19.
2001-06-19	Uwe Kühn	2.00.00	-	red.	First formulation as print document.

1 Motivation

With constant increase of multiplex applications new interfaces are created for data exchange between multiple units. The question about the necessity of new interface definitions is not relevant when one or more units already exist and the engineer has an order to create another unit which has to adapt itself into the existing interface. This situation, in which the engineer and his colleagues find themselves, is similar all over the world.

First problem: During development no real unit is available to communicate with other units (because the unit is still under development)

Second problem: The Electronic Reliability Laboratory (ERL) department tests the unit for faults so that it can go into production. Because it does not undergo a black-box-test another test should be held under certain limits.

Third problem: A tool is required to test the interfaces, but even here, other units cannot be used as test equipment as they are not able to perform the function which the interface testers want them to. Beyond that it needs an effective tool for the actual test-stand controlling.

Fourth problem: Microcontroller applications in particular are efficient in the fact that they can be adapted to the different targeted applications (intended use) by simple parameterisation, which quite frequently is realized over the existing (diagnostic) interfaces, for which then yet another independent tool is needed.

Fifth but not yet last problem: The service technician in the field needs an effective tool for error diagnosis and if necessary for the re-parameterisation of the equipment, again over the interfaces mentioned before.

So much to the situation from the viewpoint of the interface problems.

Let's regard the expectations of the different users to such tools. The service technician needs a software, which helps him to support his technical understanding of the device to be examined; a software with technical level, that however is used by him solely in the role of a user.

The process engineering in the factory must be able to build an application software with which also unskilled staff at the assembly line can get along without any difficulties.

The technical testers must create an aid for themselves, which allows them to determine the limitations of the technical abilities of the test specimen.

And at the beginning of the chain the developer needs an effective tool with which he can verify his software quickly and easily.

Concerning the used computer platforms a rather non-uniform situation exists, as well. While the developer usually works with a PC, this is not necessarily granted for the

process engineering, which more likely perhaps uses Unix workstations, and the service technician either has an industry PC in its workshop or even a notebook computer in the service car.

Remote maintenance constitutes an additional problem. It is most desirable that for the examination and solution of difficult problems the developer does not necessarily have to be sent on journeys, if as well he possibly could solve the problem from the distance. Which results in the necessity for universal interfaces over remote connections (as for example the Internet).

And then there remains still another problem. If all aforementioned problems should one day be solved for the current device generation, immediately the question will arise to what extent those solutions are reusable for the next generation.

When reviewing the different problems listed above, it can be at least inferred as a common characteristic that in the long run all involved parties have to deal with the same device and the same interfaces. Obviously soon the question arises why their problems should not be solvable with one and the same tool. The advantage would be overwhelming, since the interface problems would have to be solved only once and the different applications could be build from a mutual pool of partial solutions in a kind of interplay.

Beyond that the question comes up whether for example the developer would be lucky to train himself for each new project into a new tool as it unfortunately is common today with integrated development environments, compilers and emulators. And taking these considerations still further another step: The service technician surely will want to service more then one device generations with the same tool, since otherwise he had to face the necessity to buy a new measuring and test equipment for each following device generation.

The concept of AIDA takes all those issues into consideration and for its realization loans on existing solutions were attentively taken and thoroughly assembled to form an integrated whole.

2 Structure of the AIDA System

The following chart illustrates the fundamental structure of the AIDA System:

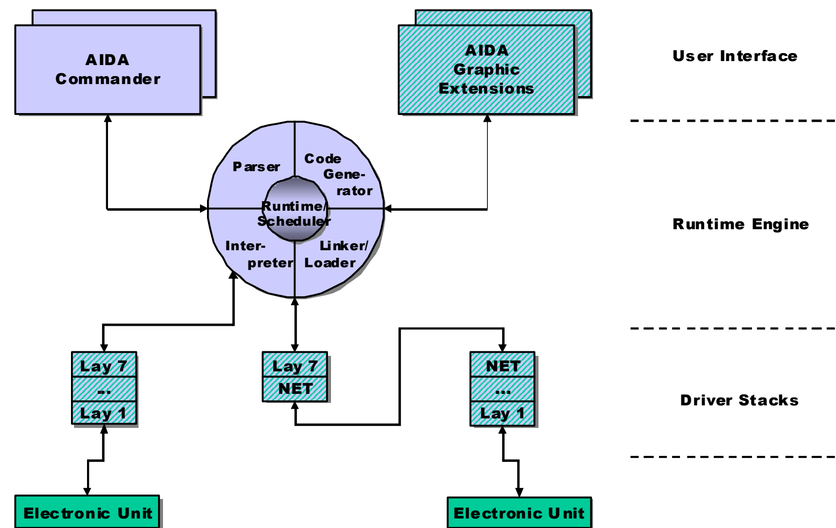


Fig. AIDA System Concept

For the ambitious developer the system communicates with the user over one or more instances of the AIDA Commander. In this case it will be a graphic but text-based in/out window, somewhat similar to a terminal window. But the difference between the AIDA Commander and a terminal window is that the AIDA Commander has additional functions like an object browser, soft switch buttons, the flexible link to choose different source-code editors, window debugging and lots more. Because the AIDA Commander communicates via the standard network interface the system's remote control is not a special option, but a targeted functionality.

Apart from the standard input/output in the AIDA Commander the AIDA System can use arbitrary other i/o channels. Usually, if not straight alphanumeric i/o is more favourable, information is directed into graphic i/o objects. These "Visual Objects" within the graphics extension are likewise linked up over network interfaces, so that i/o is easily possible on distant computers. Besides, the kind of the used graphic objects is limited, so that also extraneous graphical objects can be integrated and used, as long as they extend the basic class interface and are addressable over a network interface. Thus, users have the possibility to develop their own graphic objects with a system of their choice and appropriate to their needs and to link those up to the AIDA System.

The interface concept of the AIDA system provides driver stacks, which are designed in such a form that at runtime driver components can be loaded and parameterized from within the system. For the driver components a uniform construction specification is published, so that also extraneous components, which were provided according to this

concept, are executable within the AIDA system. The fundamental separation between interface and application is crucial, so that reusability is ensured for all the driver components which were once provided.

The core constituent of AIDA is the event-controlled multi-threadable AIDA Runtime System. It contains the P-Code Interpreter, which executes the code of POOL modules built from compiled and linked source text lines as well as user-interactive POOL command line instructions. Furthermore, the integrated linker/loader is always active, which makes it possible to load further modules at runtime. Moreover, with the AIDA Runtime System it is also possible to execute low-privileged applications specifically developed for end users.

For the developer of AIDA applications, there exists the developer version which in addition also contains the POOL Compiler. Thus the separation from privileged and non-privileged users is possible in an informal way. However, even for the owner of a developer version it is impossible to regenerate the source code from the compiled code, which guarantees investment protection for customer created applications/libraries in the AIDA System.

3 POOL

The AIDA System provides and uses its own programming language named POOL. Single commands on command level can be executed as well as entire module or function libraries. I.e. POOL is the integrated programming language of the AIDA System.

There are already so many programming languages, why thus a new one, POOL?

In the AIDA System two basic demands face each other, which do not necessarily go together very well. On the one hand, for the command line input a to some extent comfortable and efficient scripting language is needed, with which one can set off an entire sequence of commands as well. On the other hand, command sequences shall be combined into fixed modules or libraries and therefore exhibit the quality of a solid programming language. From the beginning, in order to keep the system simple, it was one of the basic demands that in the AIDA system there should not be used two different kinds of languages for the controlling of the system. During the investigation into existing approaches for this purpose it became soon evident that there were so far neither a scripting language, which kept up with serious, complex programming requirements nor a programming language, that was really interactive.

So, the decision fell at last on the language POOL, which contains the desired and necessary characteristics:

Pool is easy to learn

Due to the very clear semantics and the strict syntax of Pascal, POOL was designed on the basis of this language. Pascal itself was originally created as a training language (by Niklaus Wirth at the ETH Zurich in Switzerland).

POOL is flexible

Despite its derivation from Pascal, committed to the name POOL, many loans from other programming languages were inferred and included into the "pool" of possibilities.

Pool is object orientated

As language scope serves Borland's extension to Pascal as an object-oriented language. This is obvious, since it is of advantage to consider all the parameters and measured variables, which such a system has to handle, as objects with properties and methods.

POOL is portable

The entire POOL System is written in ANSI-C, just like the parser, the code generator and the code interpreter, and is therefore in its whole easily portable to other operating systems.

POOL is system independent

POOL uses uniform data types on all machines. The code produced by POOL is that for a virtual machine and therefore executable on really all platforms. Since on the level of the virtual machine all object references are dissolved, it therefore can be executed on

small computers, as for example intelligent interface boxes, as long as they provide a small code interpreter.

POOL is open

C-function calls are used as interfaces to general function libraries and operating system calls. Thus the developer is in the position to integrate existing libraries into the AIDA System. Beyond that, this simplifies the possibility to execute/trigger system calls, since API calls are usually provided as C-calls.

POOL is transparent

The code produced by POOL contains debugging information, which can be used for genuine source code debugging.

POOL code is protected

The code produced by POOL is for a virtual machine and therefore just as protected from re-engineering as compiled code.

POOL is a highly efficient command language.

Since POOL supports runtime type information, also complex statements are realizable as commands. Thus a highly flexible command syntax is achieved, without giving up the basic strictness in the design of the syntax of POOL.

4 AIDA Driver-Stacks

4.1 Basics

Since 1983 the OSI reference model exists for interfaces, which is also called the layer model. It was issued by the Open Systems Interconnection working group, which was brought into being in 1977 by the International Standardisation Organisation (ISO) with the goal of describing a standardizable structure of (tele-)communication paths. It is remarkable, that in this case not as usual an existing standard was declared as the world standard, but that instead a standard developed on a theoretical approach was provided before the development of appropriate interfaces. Today this standard is (despite critics in special points, in particular because of the strong orientation on telecommunications) in its fundaments accepted to a large extent and, not least on pressure of influential institutions and authorities, it is also actually implemented. Its substantial advantage for an implementation outside the telecommunications sector consists in the designation and definition of layers, whose translation to general interfaces is possible.

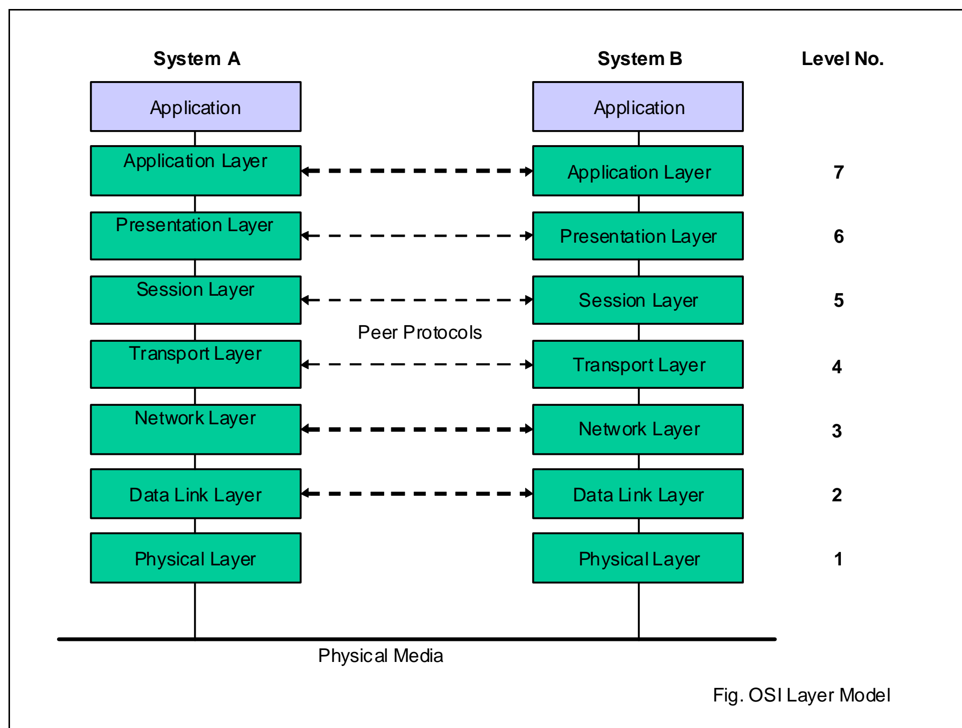


Fig. OSI Layer Model

As much as the layer model is accepted and spread throughout the telecommunications and computer-networking community, as little known however it seems to be in other fields of industrial applications. Only so it is explainable that many industrial interfaces are designed in such a way that they cannot be directly represented by the layer model.

The usual design defects here lie in the gluing of layers and crossings in the layer structure. When embedding old protocols into new protocols it even frequently comes to nesting. All too often developers feel qualified to define a new interface, possibly assessed optimal for their specific task and considered the actual problem as solved, if telegrams can be exchanged between the devices involved. Unfortunately that is usually not even half of the truth.

Since the task of AIDA does not lie in the strict promotion of the layer model, but in the realization of existing interfaces, which are possibly not conform to the layer model, compromises must be accepted for the setting-up of an interface driver reaching up to the application. In order the fundamental goal of being able to combine driver layers with one another remains realizable (only in such a way the expenditures raised for their developments can be reused profitably), inevitably layers, as far as they are inseparable or exhibit crossovers, must be combined. Ultimately for the application from the combination of driver components always a driver stack must be created, which implements all necessary layers up to the application level.

4.2 The AIDA Interface Driver Concept

The driver concept chosen for the AIDA System represents the consistent transfer of the requirements formulated in advance with an as close as possible adherence to the OSI layer model.

In the ideal implementation of the OSI reference model the interface path from the application to the electronic counterpart station was developed layer by layer. The reality usually looks different, as most protocols omit layers (usually some are actually not needed outside of the telecommunications area), or, as previously mentioned, exhibit various oddities. The AIDA system here demands interface components, which, one constructing on the other, result in a driver stack, which implements the interface completely. For this the involved components must satisfy three fundamental demands:

- A public definition section of the components permits a general examination by the system, to what extent interface components can be constructed one on the other.
- All components must support multi-threading.
- The configuration of the components is made top down by the application.

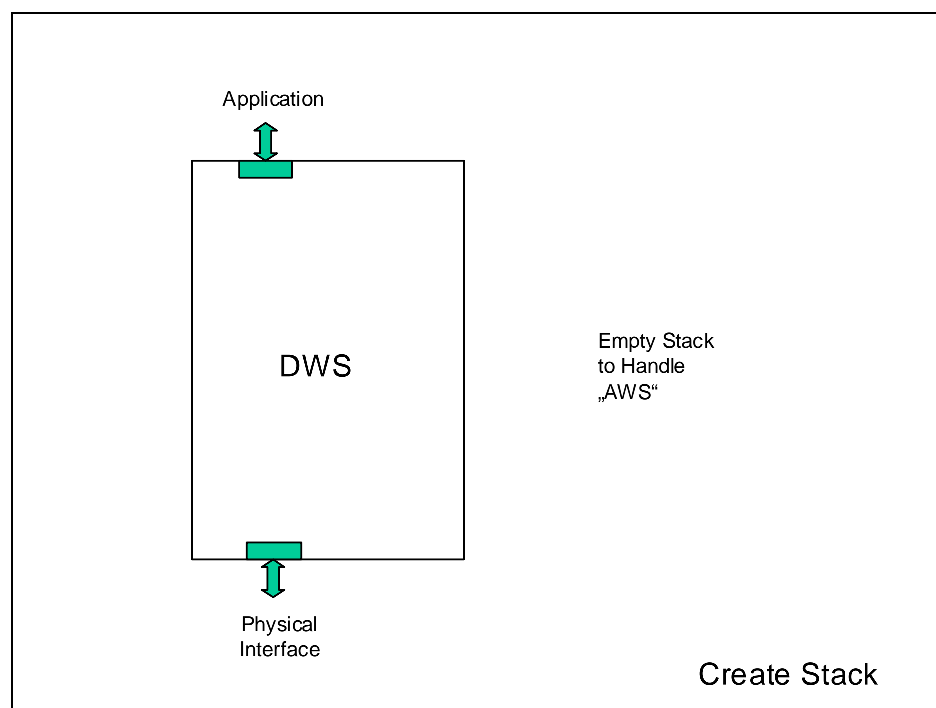
The first demand prevents from incompatible components assembled to a stack in a way that it could be discover only at runtime that the desired data path was not establishable. For that purpose the public definition section fulfils a key-lock-function, so to speak.

With the second demand it is achieved that each component retains control of the interface, even if two or more data paths are established over the same interface with possibly different parameterization (for example a different word format or transmission rate).

The third demand permits the parameterization of individual components by the application without the necessity for a separate tool to intervene in the configuration, which is usually normal with operating systems. This demand is reinforced by the possibility of the reconfiguration of an interface in conformity with current data transmission, which would not be possible in the case of an external parameterization.

This will be illustrated by the following example (in POOL), where a driver stack with BSK diagnosis protocol over a serial interface will be set up. For this purpose only two components are needed, which are available within a PC environment in each case as DLL. Since AIDA must be able to serve several interfaces at the same time, first a handle is assigned, which from now on clearly identifies the interface to the selected device, here for example a distance warning system (DWS).

```
var
  hDWS: tHandle;
...
  hDWS := AIDA_hCreateStack;
```

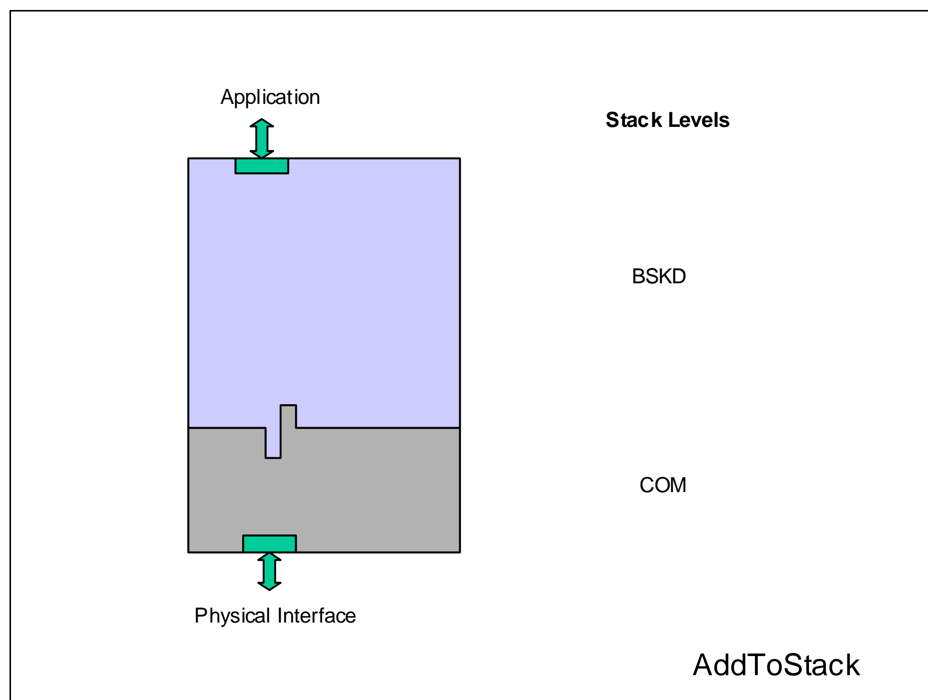


Afterwards the driver stack is set up, however - and that is important - beginning from the topmost levels. Thus it is ensured that the connection of the interface to the application remains intact at any time.

```
type
  tStackLevel = Byte;
var
  xoBSKD, xoCOM: tStackLevel;
...
  xoBSKD := AIDA_bAddToStack (hDWS, "BSKD.component");
```

```
  xoCOM := AIDA_bAddToStack (hDWS, "COM.component");
```

Here the function AIDA_bAddToStack returns (with the set-up of the stack) either an arbitrarily assigned stack level, which identifies from now on the protocol layer within the stack, or otherwise an error value, if the addition of the component was not possible.



The following function calls will parameterize the components, which are addressed by their stack levels:

```
var
  boReply: Boolean;
...
  boReply := AIDA_boSetStackParam (hDWS, xoCOM, "COM", 1);
  AIDA_vSetStackParam (hDWS, xoCOM, "Baud", 38400);
  AIDA_vSetStackParam (hDWS, xoCOM, "Format", "8E1");
  AIDA_vSetStackParam (hDWS, xoBSKD, "Retry", 2);
  boReply := AIDA_boSetStackParam (hDWS, xoBSKD, "FIFO", 14);
```

In each case the parameterization function returns a value, which states whether the parameterization was successful or not. In accordance with POOL guidelines the function can be used as well as a procedure (without the inquiry of the return value), if it is beyond question that the parameterization can take place (the subsequent three parameterizations will demonstrate this). The last one of the five instructions is unsuccessful, since the parameter "FIFO" does not match the parameter set of the BSKD component but instead of the COM component. This parameterization error is noted accordingly as the boReply value. You recognize how the use of the Stack level variables clearly assigns the parameterization to the corresponding driver component.

For the keywords the parameterization always uses the String type. The values are handed over in their natural form as String, Double/Real64, LongInt/Int32, LongWord/DWord or even as pointer. In this context it is guaranteed by the NETClient component that the parameterization is possible also beyond networks and different operating and processor systems (details are described down below). Differently than in the example given above, as the recommended working style the parameterization of a driver component should take place immediately after binding to the stack, since parameters are often already needed for configuration when the next component is added to the stack.

In order to be able to query individual parameters another function is needed:

```
{ AIDA_tstParam: Element of an AIDA-Parameter list (see aida.pli.pli) }
type
  AIDA_tpstParam = ^AIDA_tstParam;
  AIDA_tstParam = record
    phsParamName:    tpHString;  { Name of the Parameter or nil for
                                   termination of array }
    bB3,bB2,bB1:     Byte;
    enParamType:     AIDA_tenParamType;
    bB7,bB6,bB5:     Byte;
    enParamAttrib:   AIDA_tenParamAttrib;
    bParamFlags:     Byte;       { Parameter flags, see AIDA_nParam* }
    bB8,bB9:         Byte;       { Reserve (in Components as bCacheInfo) }
    bVisualization:  Byte;       { Deflt. display mode, see AIDA_nPrefer* }
    unParamVal:      AIDA_tunParamVal;
    pstValListEntry: AIDA_tpstValListEntry;
  end;

var
  pstParam: AIDA_tpstParam;

...

  pstParam := AIDA_pstGetStackParam (hDWS, xoCOM, "Baud");
  pstParam := AIDA_pstGetStackParam (hDWS, xoBSKD, "");
```

The first call returns a pointer to the structure, which describes the queried parameter and its adjusted value. The structure contains a pointer to the string with the keyword as well as a type information of the parameter value and a pointer to the parameter value itself. Within the driver component this structure is embedded in an open array, which ends, if the pointer to the keyword String holds the value nil. Consequently the function returns a pointer to the first element of the open array, if no search string was handed over, and the value nil, if the keyword searched for is not an element of the list. In this way a particular parameter or all parameters of a driver component can be determined.

With the function AIDA_boRemoveFromStack the driver stack can be gradually dissolved again:

```
  boReply := AIDA_boRemoveFromStack (hDWS, xoCOM);
```

Again the return value indicates whether the operation was successful or not. Exactly as customary in other stacks, a driver component cannot be removed from the middle of the driver chain, since otherwise afterwards incompatible components would reside one

on the other. For the sake of simplicity, in addition the entire driver stack can be easily destroyed with a single function call:

```
boReply := AIDA_boDeleteStack (hDWS);
```

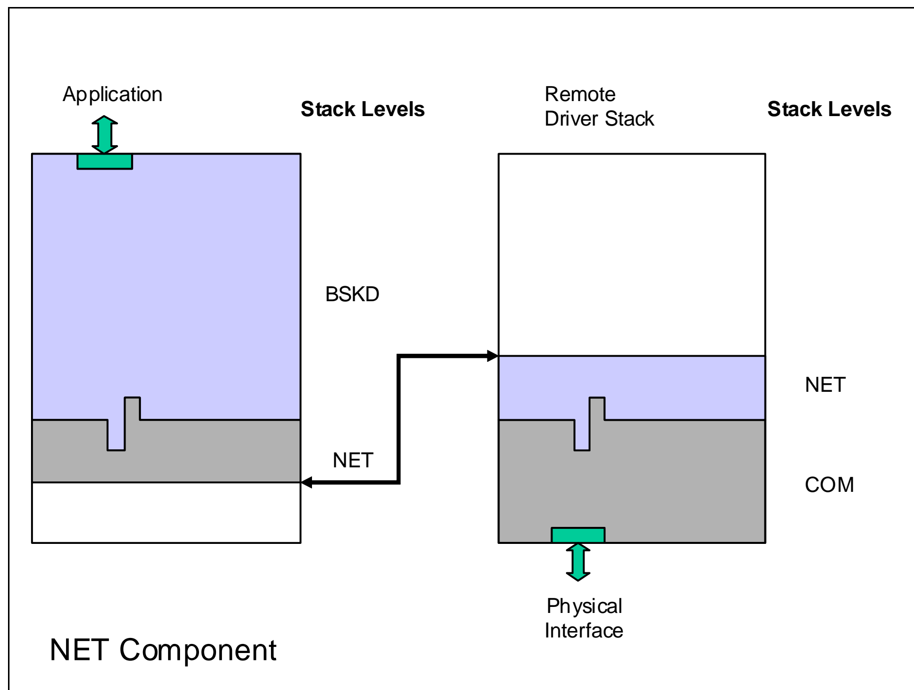
It is obvious that for this task only the handle is needed as parameter.

The NETClient component represents a completely transparent driver component. It can be inserted in either place of the driver stack (it shall not be dicussed here, to what extent this would be really reasonable and favorable). It permits the construction of a driver stack beyond the boundaries of the local computer. Analogous to the preceding example the corresponding POOL code will be something like:

```
xoBSKD := AIDA_bAddToStack (hDWS, "BSKD.component");
xoNETClient := AIDA_bAddToStack (hDWS, "NETClient.component");
boReply := AIDA_boSetStackParam (hDWS, xoNETClient, "RemoteAddress", "BSK-WST-030");
```

Notice, how striking simple the set-up of a complete interface to an foreign computer is; it is sufficient to insert the NETClient component and to establish the connection. In the first place, the addition of the NETClient component is always successful, because as an intermediate layer it is compatible to all driver components. Contrary to the first example, this time however the NETClient component must be parameterized immediately in any case, because no further driver stack component can be set-up without this. Setting of the RemoteAddress is only successful, if the AIDA server service is installed and was started on the remote computer. If the connection over the NETClient component once is established, the further set-up (in the example the COM component) of the driver stack takes place like before and with the same commands, now however already on the remote computer.

```
xoCOM := AIDA_boAddToStack (hDWS, "COM.component");
```



The following sequence copied from the preceding example

```
boReply := AIDA_boSetStackParam (hDWS, xoCOM, "COM", 1);
AIDA_vSetStackParam (hDWS, xoCOM, "Baud", 38400);
AIDA_vSetStackParam (hDWS, xoCOM, "Format", "8E1");
```

now, of course, parameterizes the COM interface on the remote computer! After the connection is established, here the possibly different parameter formats are converted by the NETClient components running on the both computers, if necessary. It is not necessary for the users application to know this!

4.3 The Structure of AIDA-Drivers

Since an AIDA driver stack resides between the application and the physical interface (resp. the driver provided by the operating system of the target platform), three different kinds of driver components are needed:

- At the top of the stack the component exists, which provides the API for the application. This topmost component is a fixed part of the AIDA system.
- At the bottom of the stack there is a pure interface driver. This component has the task to abstract the different APIs of different OS-specific interface drivers and to provide a standardized API. A basic set of interface drivers is provided with the AIDA system (e.g. for the control of serial interfaces named "COM").

- Between these two kinds of driver components different transportation protocol components can be bound (e.g. for BSK diagnosis).

In order to be able to optimally use the features of the respective operating system and to reach a high execution speed of the modules, all driver components are implemented in ANSI C. On the Win32 platform these components are available in the form of DLLs.

Each component has two public info blocks, which define to what extent two components can be bound together when a stack is about to be assembled. The info block is fixed toward the application level of the stack (upward, "lock") in each case, however, the info block to the next lower partner ("key") can vary depending on the adjusted parameters of the driver component. Therefore an application should generally proceed with the assembly of a stack in such a way that a component is parameterized immediately after binding to the stack, before the next component is loaded.

Furthermore each component provides a public parameters list for its configuration. By means of the appropriate API functions all parameters supported by the respective component as well as their ranges of values can be obtained, resp. the values of individual parameters can be changed. A component residing nearer to the top of the stack has the capability of filtering values. If a certain parameter of a component residing lower in the stack it is mandatory for the own parameterization of a higher component and therefore must not be changed by the application, this driver component closer to the application can hide the concerned parameter of the lower component from the application.

4.4 The API of the AIDA Interface Drivers

In a Win32 environment the API functions process both ASCII and UNICODE characters, if needed. However, the mixed use is neither intended nor supported, i.e. if UNICODE has been defined, it is mandatory to hand over strings always as UNICODE strings.

In case of errors the error cause can generally be determined by the general Windows function GetLastError(). Caution: If no error occurs, LastError is not preset, unless explicitly described differently within the respective API documentations, and therefore the function does not return a valid value.

The nomenclature for data structures and functions is in analogy with the Windows system. Thus type definitions or #defines are always written blocked.

Beyond that, variables receive the prefixes u, s, b, w, dw, to i8, i16, i32 for union, string, byte, word, double word as well as integer values from 8, 16 and/or 32 bits width. For structures the prefix st is used. With arrays the additional prefix a is placed in front, with pointers p.

For creation and parameterization of a Stack only a few function calls are needed (here again the calls which were already presented above, now however in C-syntax):

```
Handle AIDA_hCreateStack ( void );
```

Produces a new stack.

```
Bool AIDA_boDeleteStack ( Handle hStack );
```

Deletes a stack. In addition all bound components are unloaded.

```
AIDA_StackLevel AIDA_bAddToStack ( Handle hStack, char *sName );
```

Binds a new driver component to the bottom of the stack. The returned stack level is needed in order to address a certain driver.

```
Bool AIDA_boRemoveFromStack ( Handle hStack, AIDA_StackLevel dwLevel );
```

Removes a component, as well as all others underneath, from the stack.

```
Bool AIDA_boSetStackParam ( Handle hStack, AIDA_StackLevel dwLevel,
                            char *sParamName, ... );
```

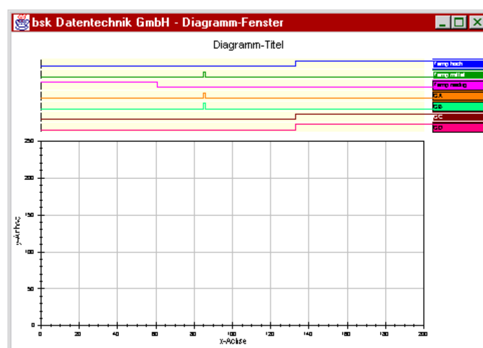
Sets a Parameter. For portability reasons the function was designed as a function with a variable number of parameters; however at present in the current version only exactly one additional parameter is expected and evaluated.

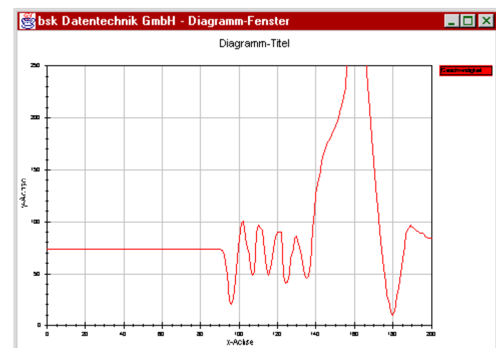
```
AIDA_Param* AIDA_pstGetStackParam ( Handle hStack, AIDA_StackLevel dwLevel,
                                    char *sParamName );
```

Determines a parameter (if `sParamName! = null`) or the list of all Parameters (`sParamName == null`).

5 AIDA Visual Objects

Whereas direct interactions of the developer with the AIDA Runtime System are predominantly carried out using the command window, real applications will rather address graphic output objects. The graphics extension constitutes an up-to-date form for the representation of measurement values, parameters, characteristics diagrams etc. Regard the following examples:





While the AIDA Runtime System forms a more or less self-contained task, graphic objects from their necessary number can quickly exceed a graspable extent. Therefore an attempt was not even started to include graphic objects into the basic inventory of the AIDA system. Instead, graphic objects are created, parameterized and displayed over a standard network interface. This method emphasizes the AIDA system's underlying philosophy of open interfaces. As a welcome side effect graphic output and the AIDA Runtime are easily distributable to different computers, e.g. to display measured values on a personal computer, while the actual test sequence takes place on a remote computer somewhere in the network.

The substantial argument for the consequent separation of the graphics extension from the AIDA Runtime System, however, is the one mentioned before. Thus it is left to users equipped with the appropriate know-how, to what extent the provided selection of graphic objects is sufficient for their requirements, or whether they extend the existing objects set by some self-defined objects.

While at the selection of the scripting language of the AIDA Runtime System none of the established languages was applicable, with the graphic objects the choice fell on the

at present most efficient language for visualized object-oriented applications, i.e. Java. Here again, the binding of the graphics extension over network interfaces does prove as a favourable choice, because the complete separation of the two systems despite different development environments, here POOL, there Java, doesn't constitute any violation of the conception.

A further substantial argument for the use of Java is its platform independence, so that all graphic objects, provided that they were created within an authorized Java development environment, should be transferable to other operating systems or computer platforms without any adaptations. The choice of the means here again excellently fits to the set of basic demands raised for the AIDA System.

Due to the perfect separation of AIDA Runtime and graphics extension the individual graphic objects are not called directly via POOL constructs. Instead, the graphic object's owned string based commands are assembled by POOL and transmitted over the interface. In the POOL standard library module VOL (Visual Objects Library) corresponding POOL objects are defined for all graphic objects.

Because of the fundamental independence of the graphics extension the information for the usage and extension of the objects as well as the objects themselves are described in a separate document.

The graphics extension contains the following configurable basis objects:

- Round Displays
- Buttons
- x/y Plotters
- y/t Plotters
- Text Windows, single- and multi-line
- Log Windows
- Dot-Matrix
- 7-Segment
- Icons, free images
- LEDs, lights
- Background
- Bar Graph
- Round Switches, digitally
- Potentiometers
- Sliders
- Switches
- Menus
- Radio Buttons
- List Boxes
- Grouping Elements
- Input Line
- File Dialog Window

Owing to class inheritance under Java further objects can easily be derived from the classes of the basis objects.

Common for all graphic objects is their free positioning within the graphics window. Full-graphic elements such as icons, backgrounds etc. must be made available in the file system from where the graphics extension is started.

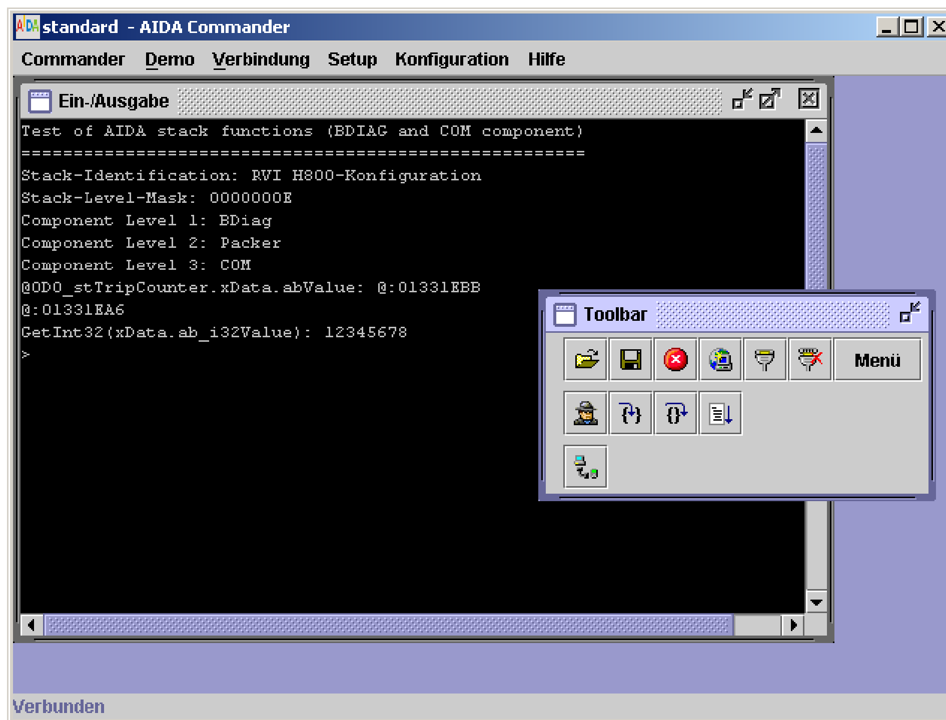
The graphics extension is connected to the AIDA Runtime in such a way that an event control is possible, i.e. the events released by the graphic objects are fed into the event control mechanism on the (POOL-)side of the Runtime. Thus apart from the command transfer and the event transfer no other data traffic occurs on the network interface.

Independently of the abilities of the AIDA Commander, complex graphical user interfaces (GUIs) can be created with the graphics extension, however, unlike the Commander these surfaces cannot directly interfere in the runtime system.

Thanks to the workability of objects under Java, classes of basis objects are easier to deduce further objects.

6 AIDA Commander

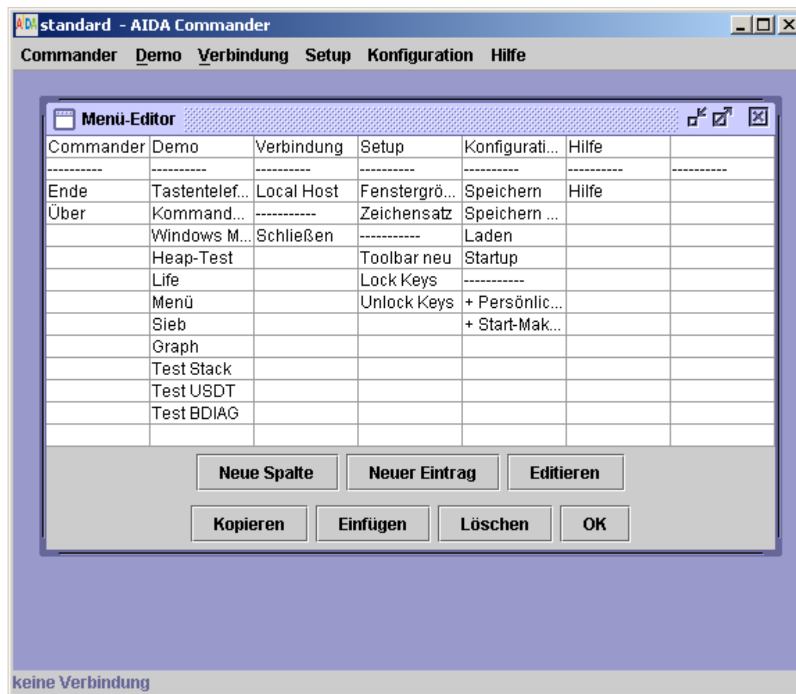
The AIDA Commander serves for the direct interaction of the user with the runtime system. At a first glance the Commander appears to the AIDA developer like a pure terminal window and as such it can also be used, as in the command line individual commands are entered and transferred to the AIDA Runtime System for execution and the output of the AIDA Runtime is also displayed again in the command window.



Actually, commands are handled on a level comparably to a terminal application. In principle thereby the AIDA Runtime is completely controllable by (command line) commands. This kind of (consistent) separation between runtime system and command window emphasizes again the AIDA System's underlying philosophy of the system’s flexibility, so that also the complete user front-end can be spatially separated from the runtime system (for example the runtime could run on a remote computer in Australia, while commands were given across the Internet from Germany).

6.1 Interactive Configuration

The surface of the AIDA Commander can be freely configured interactively. For that purpose menus and tool button bars are available. Together with the runtime, menus and buttons can be disabled/enabled depending on the current context.



Like the interactive build up of menus and buttons, the events carried out can be freely configured, as well.

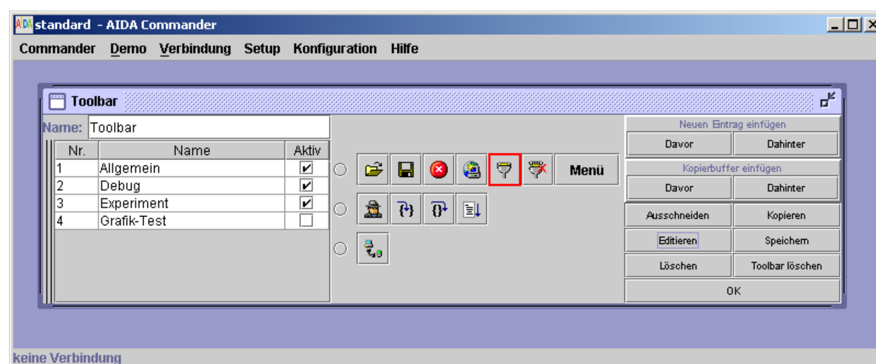
When setting-up commands it must be differentiated between commands for the AIDA (POOL) application, which can also consist of several lines, and such for the control of

the runtime itself, here called "internal" commands. Internal commands are marked by the sequence <IC> and these serve for

- the establishment of the connection to the AIDA runtime,
- the configuration, e.g. character set adjustment, of the Commander
- transmission of debug instructions to the AIDA runtime,
- the single step control of debugging operations,
- the query of variables, objects, constants
- or the execution of external programs.

A shortcut (from a combination of keys) can be assigned to each menu option. The menu entry can be named with a free text.

In the same way buttons are configured, whereby these can alternatively be provided with a graphical symbol. In addition the button width can be varied.



Both buttons and menu options can be defined as "executable" or "insertable". In the first case the assigned instructions are directly executed, in the second case they are just inserted into the command line for combination with further instructions. Thus the ability of POOL to set off whole command sequences is supported, e.g.:

```
repeat <MenuEntry> endrep
```

"Executable" is used as the default option, but for convenience the "insertable" option is selectable as well by pressing the SHIFT key in addition, so that both functionalities are available at the same time.

As nowadays usual, the tool bars can be “torn off", i.e. positioned freely on the desktop.

Since both the buttons and the menu options represent only catchwords, which are not always understandable for a newcomer, the AIDA Commander permits adding of tooltips, which as usual will appear after a few seconds, if the user remains over such a control with the mouse cursor.

For really hard-boiled command line users menus and buttons can also be called over ALT-codes. Thus the AIDA System replies to the entitled reproach that it is uncomfortable and inefficient to have to use the mouse for a few operations while intensively using the command line.

One of the adjustment possibilities concerns the character set of the command window. AIDA permits the use of existing character sets and thereby meets all language's and country's demands.

In such a way interactively composed user interfaces can be saved into configuration files and of course also loaded again. To prevent from any abuse with privileged functions, all configuration files can be protected with a password. In this way both the personalized or project-related operation of the surface is ensured and the possibility of making reduced Commander surfaces available to not-privileged customers or co-workers.

6.2 Application Controlled Configuration

For the comfort functions specified above, the AIDA command window uses a second network channel, over which AIDA specific information is exchanged. Thus the AIDA Commander is extensible to a complete integrated development environment (IDE), without having to give up the conceptional separation from the runtime. The comfort functions are supported on the part of the AIDA runtime by a set of instructions, so that the command window can be extended to an event-controlled surface, with which one can also serve unskilled users. Independently from the interactive surface configuration discussed above, the AIDA Commander possesses all abilities to control the configuration of the surface from the AIDA runtime. Comfort functions, supported by the Commander, are:

6.2.1 Instructions for the Creation and Parameterization of Windows

Thus full-graphic control panels can be made available apart of the actual command window.

6.2.2 Instructions for the Set-up and Execution of Menus and Submenus

With these an intuitive control panel can be made available also to unskilled users (e.g.: Selection of parameters from groups of parameters)

6.2.3 Instructions for the Creation of Dialog Boxes with Related Controls.

Thus interfaces or parameters of controllers can be parameterized in the common Windows control philosophy.

6.2.4 Instructions for the Creation of Toolbars

In this way tool bars can be provided and self-defined functions can be configured for each button.

6.3 Comfort Functions

Substantial comfort functions, as were contained in earlier BSK systems already, have been adopted.

Thus the AIDA System keeps its own command stack, which facilitates recalls of commands already executed. Each Commander instance manages its own command stack; in addition specific stacks can be used within each command window. An example of this is the interactive input of file names, which does not have to do anything with the other commands already handed over. The administration of the stacks can also take place within POOL procedures; an important reason for their administration within the runtime.

Since the output lines normally run quite fast out of the visible range within the command window, the AIDA Commander permits the adjustment of a virtual terminal window, which goes far beyond the normal dimensions. The visible range of the window can be freely adjusted within the boundaries of 192 characters in x-direction and 144 lines in y-direction, whereby the individual defaults can be combined with a suitable character set. These boundaries apply also to the positioning functions from within POOL applications.

6.4 Development Devices

For the development of applications the usage of suitable editors and browsers is essential. The AIDA System makes it possible to the user to use the usual tools he is familiar with. BSK recommends lean text editors like PSPad or TextPad, for which also a syntax highlighting configuration files for the POOL language are provided. There is also a full featured IDE in preparation named “AIDA Studio” basing on the Eclipse framework.

By the employment of integrable tools it is possible for the developer not only to see source code but also to change it interactively. The AIDA Runtime takes care for a seamless integration of the changed source codes and also supervises the access to common resources.

The call of external programs as well as their linkage to buttons or menu entries is generally possible, whereby internal system variables in the kind of environment variables can be inserted into the calling parameters.

6.5 Help System

AIDA offers on-line assistance on the basis of HTML files. With the call of the help function the user's primary HTML browser is launched. With conventional tools for the production and editing of HTML pages thereby the user can provide own assistance systems without any troubles. Also POOL and C-libraries can be documented easily in this way. A strict separation between the help system of the AIDA Runtime itself and that of an AIDA application does not exist. Thus it is left to the developer's decision, what kind and to which extent he wants to provide assistance to the user of an AIDA application.

6.6 Debugging

POOL permits source code debugging on an intermediate code level, since this level has been designed for the inclusion of such information. The AIDA Runtime uses these information for insertion of breakpoints and for the single-step execution of POOL statements.

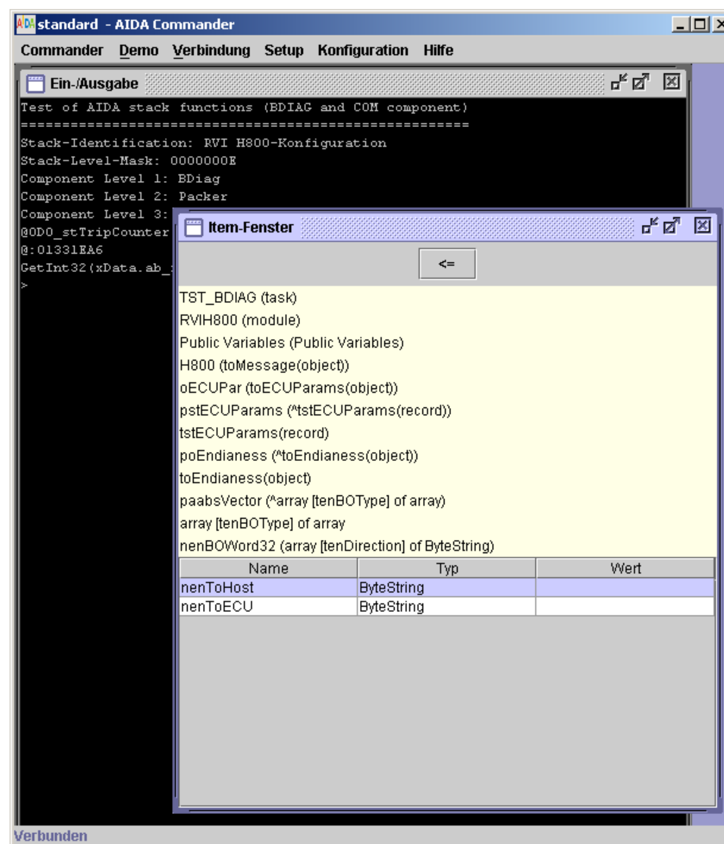
Breakpoints can be inserted in arbitrary place in the POOL code. All defined breakpoints are managed in a common list and can there be viewed, individually or entirely activated and deactivated.

For internal use the AIDA System will permit also single steps on intermediate code level. However, this feature is not part of the distribution.

6.7 Data Objects

The AIDA Commander supports on-line administration of all active global data objects. Such data objects are data areas of objects (member variables), global variables and global constants within the respectively valid context.

Data objects can be viewed and selected within hierarchical lists. Thereby the selection can be limited by a search mask. The hierarchy of the data structures is supported by folding (opening and closing) of the levels, descriptive similarly to a hierarchical file system.

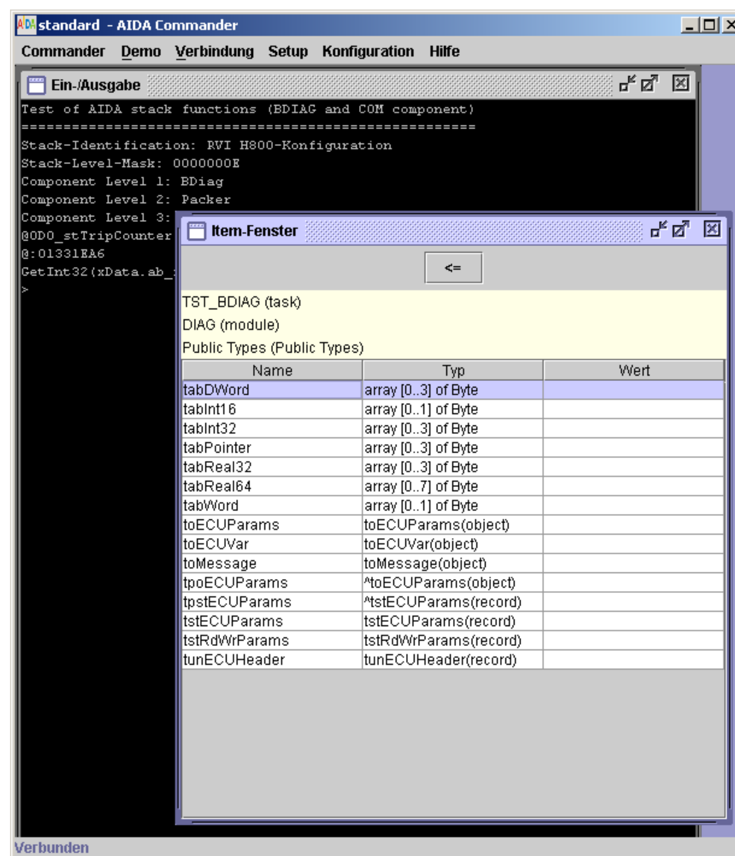


Selected data objects can be grouped together into separate watch-windows and then are subject to the automatic actualization of their contents by the Runtime System. Within the watch-windows the actualization characteristics of the observed data objects can be individually activated and deactivated. Several different watch-windows are possible at the same time.

Thus the AIDA System provides the singular possibility of seeing all data of the system in real time. These possibilities arise also from a characteristic of the POOL System, since here the type characteristics of all data types are known at run-time of the system. That is not the case with purely compiling systems.

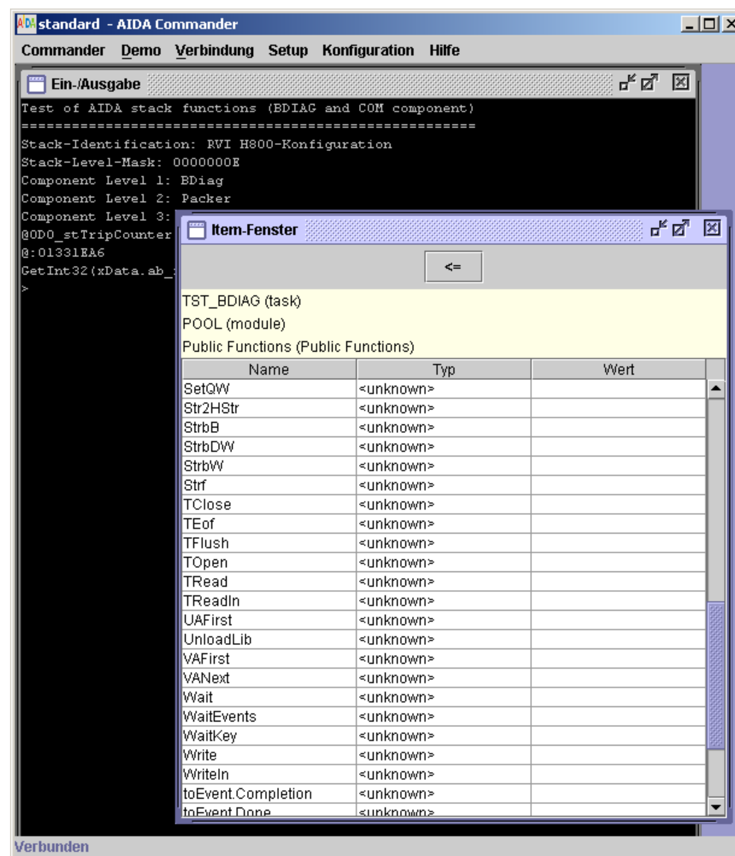
6.8 Data Types

Similar to the hierarchically representable structures of the data objects (= instances of data types) the used public data types themselves can also be browsed. The selection takes place again through hierarchically partitioned lists.



6.9 Procedures (Macros)

All procedures available in the assigned context can be browsed within a selection list hierarchically and be restricted by a filter.



The selection of the procedures from the complete list is used likewise to manage breakpoints. In a separate viewer the AIDA System permits the browsing and setting of breakpoints.

6.10 POOL-Threads

Another characteristic of the AIDA System is the possibility of splitting off further Commander windows as independent threads. Usually the command-supported controlling of the system by the developer takes place within a Commander window. The execution of a command can be started however alternatively in a separate window and continued then there. Contrary to the also possible starting of a further AIDA Commander, however, the derived window here takes over the preset characteristics of the output window and starts within the calling context. Model for this are the similar qualities of Internet browsers, which likewise permit an opening of independent windows on the same connection.

As far as technically possible, these derived windows are registered when leaving the AIDA Commander and broken off threads are taken up when the Commander is restarted again.

7 Index

C

Commander 22
Contents 2

D

Driver Stacks 10

I

Index 33

M

Modification History 3
Motivation 4

P

POOL 8

S

Structure 6

V

Visual Objects 19