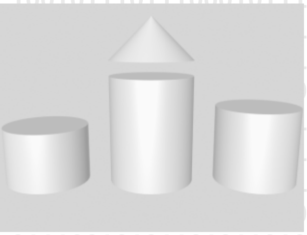

ODABA^{NG}

Function Reference

run Software-Werkstatt GmbH
Weigandufer 45
12059 Berlin

Tel:	+49 (30) 609 853 44
e-mail:	run@run-software.com
web:	www.run-software.com

Berlin, October 2012

Content

1 Introduction

DEPRECATED

The described Interface is partly deprecated. Please refer to the online documentation on http://run-software.com/content/documentation/ to review the current description. However as the SystemInterface is still in use this document can be useful.

ODABA2

ODABA2 is an object-oriented database system that allows storing objects and methods as well as causalities. As an object-oriented database, ODABA2 supports complex objects (user-defined data types), which are built on application relevant concepts.

ODABA2-applications are characterised by a high flexibility that is achieved by supporting in addition to object (concept) hierarchy, multifarious relations between objects (master and detail relations, relations between independent objects and others). This way conditions and behaviour of objects in the real world can be represented considerably better than in relational systems.

ODABA2-applications cannot only be drawn up as event-driven applications within the field of the graphical surface but also at the database level. This is one more way in which the application design is very close to the problem.

This makes ODABA2-applications a favourite possibility to solve highly complex jobs as come up in administrative and knowledge areas.

Platforms

ODABA2 supports windows platforms (Windows95/98/Me, Windows NT and Windows 2000) as well as UNIX platforms (Linux, Solaris).

You can build local applications or client server applications with a network of servers and clients.

Interfaces

ODABA2 supports several technical interfaces:

- C++, COM as application program interface (this allows e.g. using ODABA2 in VB scripts and applications)
- ODBC (for data exchange with relational databases)
- XML (as document interface as well as for data

exchange)

User Interfaces	ODABA2 provides special COM-Controls that easily allow building applications in Visual Basic. On the other hand ODABA2 provides a special ODABA2 GUI builder.

2 Class Overview

Access Classes

Acess classes are classes that provide access to the database. Access classes are provides as access class handle on different levels (client, dictionary, database, database object, collection, property).

ODABA Server

A ODABA server will manage any number of databases. After creating an ODABA server it can be started and halted using the functions Start() and Stop(). There is no login required for connecting to the server, however, for accessing a database you may have to pass login information to the server. Login-Information must be passed to the CreateClient function. You can overload this function in your application procedure to provide specific log-in checkings and other services for an application ODABA2 server.

The ODABA-server maintains a list (catalogue) for database files. This catalogue must be stored under server.ini in the ODABA2 installation path. The catalogue section starts with [ODABA-CATALOGUE].

ODABA client

To run client server applications you must create a ODABA client instance. To support several connections to different servers you can create one or more clients within your application.

When connecting to different servers you must create one client for each server. You can open several clients in an application. The first client, however, is considered to be the main client. The main client should be the last client closed in an application. After closing the main client you can open another main client. Since there is no hierarchy defined between clients the system will not check

The main client registers the process and activates the error log file. It opens the system database for providing error messages and the data catalogue if one has been specified in the system environment (see ODABAClient constructor). These information are described in an ini-file, which can be passed to the client.

For initializing and registring the process properly a client should be created also for locally running applications.

Dictionary Handle

The dictionary handle is used for providing schema definitions from the dictionary. The dictionary creates internal images from the externally stored schema definitions. These internal images (**{.r DBStructDef**}) can be provided by means of dictionary functions.

Because the dictionary is a database handle **{.r DBHandle**} you can access schema information also directly via PI functions.

Database Object handle

Database object handles are necessary for accessing data in an database object. A database object can be considered as a database within a database. Each database has a root database object on top. Below each database object any number of subordinated database objects can be created.

Database objects in a database are logically separated but not physically. Thus, it becomes possible to establish links between structure instances in different database objects. Each database object has, however, its own extents containing the global instances of the database object.

The database object handle for the root database object is part of the database handle (-> DatabaseHandle) and need not to be opened explicitly.

A database object handle is required for opening extent property handles for accessing structure instances stored in extents.

The database object handle administrates transactions. Transactions can be started and stopped for each object handle. The database object handle is not thread save, i.e. a database object handle must not be used simultaneously in different threads.

The database object supports version slices, i.e. each database object may have its own current version.

Database Handle

Database handle must be created for accessing data in an ODABA database. An ODABA database must be connected with a dictionary, which defines the object model for the database.

Each ODABA database consists of at least one Database Object (Root Object) that is the owner od extents and other data collections.

When creating a database handle the object handle this is based on a database object handle (-> DBObjectHandle) for the root object, i.e. the database handle inherits all the functionality from the database object handle.

A database may consists of a number of physical separated mainbases, sub-bases and data areas. This is, however, handles internally after creating the database. For creating a multiple resource database the database handle provides several functions for initializing main and sub bases and data areas.

Moreover, the database handle provides log and recovery features, that allow logging all changes made on the database or recovering the database in case of errors.

The workspace feature supported by the database handle is a sort of persistent transactions. It allows storing changes for a longer period outside the database and consolidating or discarding changes when requested by the user.

Property Handle

Property handle are used to handle persistent or transient **data source**. A data source is a collection, object instance or an elementary database field. A data source contains the data for a property of a specific object.

A property handle usually handles a collection of subsequent object instance. In special cases the collection is singular (e.g. the 'direction' for a persion is exactly one 'Adress' object instance). In other cases the instance is elementary (as eg the given names of a person).

A property handle has a cursor function that allows to select one of the instances in the collection as the "current" instance. Only from the selected instance you can retrieve data by means of subsequent property handles or Get-functions (GetString(), GetTime(), ...) for elementary datasources.

Generic Property handles

You can define generic property handles using the generic property handle contructor (PH(type)()). This requires that you have created a C++ header file for the referenced type. In this case you can access elementary data field in the instance directly referring to the generated class members. For references the instance contains corresponding generic property handles that you can reference by class member name as well. In this case you need not to create the property handle you want to access. This makes programming simpler but in this case you must recompile the application when changing the database structure. This is not necessary when referring to property handles hierarchies created in the appplication.

Property handle hierarchies

Property handles form a tree that defines a specific view in an application. When defining this view once the property handles cann be used as long as the application follows the defined view. When defining a property handle for "AllPersons", which is an extent in the database, you can define sub-ordinated property handles for 'name', 'children', and 'company', which refer to the persons name, its children and the company the person is working for. When selecting another person in the AllPerson property handle the datasources for 'name', 'children' and 'company' will change. This, however, is maintained automatically by the systen, i.e. when changing the selection in an upper property handle the data

Opreartion Handle

Operation handles can be used for executing operations as expressions or function calls. Usually, an operation is associated with a property handle defining the instance that is passed to the operation as calling object.

Database Query Handle

DBQuery allows defining database queries by means of an ODABA view definition. A database query usually retrieves data from the database. It is, however, also possible to update data in the database.

A database query may refer in different places to ODABA OQL expressions. ODABA-OQL is an object query language with specific ODABA extensions.

You may run a query against the complete database (global context) or in a reduced context (instance context). The context for running the query can be defined in the query. The result of the query can be printed to the console or directed to an output data source.

Instance Handle

Instance handles are used to pass and return structured database instances. Instead of an instance handle a (void *) area can be passed, that is automatically converted into an instance handle. The instance area is allocated and freed by the application.

Key Handle

Key handles are used to pass and return keys. Instead of a key handle a (char *) area can be passed, that is automatically converted into a key. The key area is allocated and freed by the application.

Event Handler Class

The Event Handler Class is a base class for supporting writing event handlers. It provides some basic functionality for setting and calling event handlers for handling server events.

You may derive your own handler classes from EventHandler to provide handler functions for server events. You may overload the handler functions InstanceEventHandler() and PropertyEventHandler() for providing your application specific event handling.

The event handler allows handling instance, property (collection) or local events. Instance and property events are client server events that are generated, when an instance or collection changes. Local events are those events, which are usually handled in the instance or property context. You may, however, set event handler for local events for a specific property handle, which allows overwriting or expanding context functions.

Event Link

This is a function link object for handling events. The function link stores a pointer to the handler class instance and the function to be called.

The following status indicators are used:

stsini - handler is active and will be executed

Property handle stack

A property handle stack allows defining a series of related property handles. A Property handle stack can be defined for a property handle and allows activating a new and saving the current handle using the Push() function and re-activating the previous handle using the Pop() function. Thus, it becomes possible, e.g. defining a sequence of subsequent selections with the possibility of going back to the prevoius level.

Data Exchange

Data source

A data source describes an ODABA data source on a certain level (Dictionary, Database, DBObject, Extent, Instance). A data source can be parametrized by means of an INI-file. The INI file contains the names for the objects on the different levels. Not specified lower levels are not opened and have to be opened in the application (e.g. when defining only dictionary and database the extent is not opened and no instance is selected), The datasource is defined as section in the INI-file starting with the [datasource name].

A data source can be directed to a server. In this case the datasource has to be opened with a connected ODABA client or the INI-file must contain a server specification. In the last case the data source connects to the server automatically when opening the data source. The connection is owned by the datasource in this case.

A data source cane be opened and closed as a whole (Open(), Close()) or separately on each definition level (Connect(), OpenDictionary(), ...).

Definition Classes

Definition classes are classes that mainly provide meta information about structures and properties. Beside providing metadata the classed allow creating internally defined structures for defining temporary object types.

Definition for the internal presentation of data structures and enumerations

Definitions for data structures are usually read from an ODABA2 dictionary. However they can be provided and filled directly in main storage. Still in this case the definition should be provided via Dictionary functions to make them available for the ODABA2 kernel.

From an ODABA2 dictionary structures are provided only, if they are marked as checked and as ready for a non test environment.

Definition for the internal presentation of property data

The internal property definition contains all information available and necessary accessing data of the property. Among basic information such as type and size it contains special ODABA2 access information such as index and base collection definitions.

Alls these information are used for reading and writing data just as to execute operations on properties (see also **{.r DBField**}).

Context Classes

Context classes allow defining specific behaviour for access objects on different levels (database, database object, structure, property). Context classes define the basic functionality for context classes on different levels. Context classes allow handling a number of internal events by overlaoding the default behaviour in the application specific context classes. This is a simple way to react on events as reading or updating an instance.

Behaviour implemented in context classes is application independent, i.e. it provides a set of basic business rules and ensures logical database consistency.

General Context Class

The general context class is a base class for all database or GUI context classes. A context usually defines a data element or a data collection in its specific context, e.g. children in the context of a person or in a list in a GUI application. In a context the behaviour of object becomes more specific, which can be expressed in a context class. Context classes have to be defined in a specified form according to the type of context to be implemented.

Contexts in an application form a hierarchy, i.e. each context object has either a parent (upper) context or is a top context (e.g. database or project context).

Each context has two status properties which reflect the current state of the context. Since context classes area created and deleted by the system the current state of a context class (as opened or closed) is not always clear for the application programmer. The process state (-> CTX_ProcessState) describes the the current state in the processing. The display state (-> CTX_DisplayState) describes the visibility of the context. For GUI context classes this is the way the associated GUI element is presented at the moment on the user interface. State properties are maintained by the system but can be retrieved by the application.

Moreover, the context class provides three user states that can be updated and retrieved by the application.

Context classes signal several events that are relevent for the specific context. Thus, context classes are typically used for handling system events as delete or inserted for database instances or lose/get focus for GUI contexts. The enumeration of events supported by a context class is defined in the cpecific context class implementation.

The context class supports the action interface, which enables functions in context classes calling actions defined in a reasource database or created internally.

Base class for data base contexts

The base class for database contexts provides some basic functionality for data base context classes. In particular the class provides most of the default event handlers that can be overloaded in specific context classes. Overloaded handlers need not to call the default handlers since nothing is done in the default handlers. Handlers for database events are usually called within internal transactions. Thus, all modifications made by the event handler are reset when the transaction fails.

For a number of database operations Not-events are generated that are called in case of an error. An error could be a database (consistency) error but the process event (before event) could have denied the operation as well.

Database Context

The database context allows defining functionality that is executed when opening or closing a database. The database context does not have a parent context.

The default database context can be overloaded by a application specific database context class.

Database Object Context

The Database Object context allows defining functionality that is executed when opening or closing a Database Object. The parent context for an object context is an object con-text (if the Database Object is not the root Database Object) or the database context (for the root Database Object).

The default database object context can be overloaded by a application specific database context class.

Structure Context

A structure context is created for each structure type. It defines the connection between the instance and the instance description. Moreover, it allows determining the active con-text hierarchy for the structure instance, i.e. the parent property/extent, the structure the parent property is defined in, the parent parent property etc. Thus, the structure context defines the context in which the object instance has been provided.

The parent context for a structure context is always a property context. This can be the property context for an extent or for another property within a structure instance.

The structure context allows handling read and updating events as well as creating or deleting events.

Property contexts

Property contexts are created for extents, references, attributes, relationships and base structures. The property context defines refers to the property instance as well as to the property definition. Moreover, it allows determining the active context hierarchy for the property, i.e. the parent structure/Database Object, the property the parent structure is accessed from, the parent parent structure etc. Thus, the property context defines the context in which the property instance has been provided.

The parent context for a property context is a structure context (when the property is part of an object instance) or a Database Object context (when the property is an extent.

The property context allows handling read and update events, validity checks and insert and remove events.

The default property context can be overloaded by a application specific property context classes.

Service Classes

Service classes provide common database independent functions that support a number of internally used object (sorted or linked lists, data convertion etc).

General Error object

The error object is used to store and pass error information to the application. Errors are identified by error class and eror number. In addition the class and function name detecting the problem and a short error explanation can be provided. Moreover, an error may include upto 6 context depending error variables that can be displayed in the error message.

Usually error messages are written to a log file (error.lst) which is stored in a folder addressed by the TRACE environment or ini-file variable. It is, however, also possible to display errors on the terminal.

Usually errors should be reset in all functions that may signal an error. Otherwise the calling function may not be able to determine whether the error signaled is an old error or has just been signaled in the called function. This strategy requires, on the other hand, that signaled errors have to be saved when other functions are called in the error handling thet might generate errors again, since those functions will reset the error. You can use the Copy() function to save the error.

The way errors are presented in the application depends on the error handler installed (ErroerHandle). Usually errors are written to the console output for console applications and shown in a message box for windows applications.

Database Error Handle

The database error handle provides extended documentation for errors detected in the system. In contrast to the basic ErrorHandle the DBErrorHandle locates signaled errors in the system or application database and provides detailled information for the error detected.

A application specific error handle can be defined and set for enabling application specific error handling (-> ErrorHandle).

Enumerations

Several system enumerations define the value sets for a number of enumerated properties as access mode, structure types etc.

Replace options

This option is used to control copy or duplicate operations for instances. The replace option is based on the existence of in instance in a collection, i.e. whether an instance with the selected sort key of the target collection does already exist in the target collection (local existence) or in one of the base collections of the target collection (global existence).

Usually, when copying referenced instances the replace option is passed to the subsequent copy operations.

Action Types

Action types allow defining different types of actions that can be called for reacting on events. Some of the action types require special runtime environments (e.g. the Window Action that requires an GUI application and will not run in a console application). Eac action can be defined by an action specific resource object in a resource database, when running a database application. Otherwise the application must provide the required run-time information for the action.

3 Alphabetic Class List

ActionType - Action Types

Action types allow defining different types of actions that can be called for reacting on events. Some of the action types require special runtime environments (e.g. the Window Action that requires an GUI application and will not run in a console application). Eac action can be defined by an action specific resource object in a resource database, when running a database application. Otherwise the application must provide the required run-time information for the action.

ACT_undefined - Action type undefined

The action cannot be executed. This option can be set to prevent the action from running or to indicate an undefined action.

ACT_Constant - Constant action

The action generates a constant value corresponding to the action name. When the first character is numeric or +/- the function returns a numeric value and a string value otherwise.

ACT_Document - Document action

A document action generates a document according to a defined document template. A document action requires a resource database that contains the document template.

Document actions can be described in an ADKA_Document resource. Running a document action requires the Document Interface (DCI), which must be installed.

ACT_Expression - Expression action

Expression actions can be defined for calling a local OQL expression or an expression defined in an OQL class. OQL expression actions require a database application. The resources for an oql-Expression can be defined in an ADKA_Expression resource object.

ACT_Function - Function Action

Function Actions can be called for functions that have been implemented as context functions. Function expressions can be called on the corresponding context clas, only. The action name corresponds to the function to be called.

ACT_Jump - Jump Action

Jump actions are used to pass control to a selected GUI element. Jum actions can be called for GUI applications only and are ignored for console applications. The action name refers to the GUI target element. The action is described by an ADKA_Action resource.

ACT_ParmAction - Parameter action

With a parameter action any type of action can be called that allows passing a number of parameters. The parameter action resource (ADKA_ParmAction) defines the parameters to be passed to the action.

ACT_Program - Program Action

A program action allows running a console or windows application. The calling specification for the program to be executed can be defined in an ADKA_Program resource.

ACT_Menu - Menu Action

A menu action creates a popup menu in a GUI environment, i.e. menu actions cannot be called in console applications. Menus have to be defined in an ADK_Menu resource. The menu actions can be described in an ADK_MenAction

ACT_Window - Window Action

A Window action allows opening a new window (dialogue or form). The window action requires a GUI environment. Window actions can be defined in an ADKA_Windows resource object.

BNFData - BNF data element

A BNF data element contains the data for a given symbol in an BNF expression. BNF data elements form a hierarchie down to symbols, which have been defined as relevant symbols.

BNFData - Constructor

i0

```
BNFData :: BNFData (BNFSymbol *bsymbol, char *string, int32 string_len )
```

bsymbol	BNF Symbol Pointer to a BNF symbol.
string	String area Pointer to the 0-terminated string area.
string_len	String length The string length defines the maximum number of characters that can be stored in the string area without counting the terminating 0. Usually this value is 1 less that the allocated string area.

i01

```
BNFData :: BNFData (BNFData *bparent )
```

i02

```
BNFData :: BNFData (const BNFData &bdata_ref )
```

CreateParser - Create ad-hoc parser

The function create an ad-hoc parser for the BNF described by the BNFData tree. The BNFData tree can be constructed from a BNF definition file or string using the BNF meta parser (BNFDefinition).

The function returns a pointer to the parser, which allows analysing syntax aexpressions according to the defined BNF.

Since the function is optimizing the priority for the parser symbols, one may list the priority list for the symbols, beginning with the symbol of highest priority. When sub-parsers are referenced, the symbols for sub-parsers are listed as well. Symbols with higher priority are analized before symbols of lower priority. This solves some of ambiguity problems, which result from the fact, that the same symbol can be used as starting symbol in different production rules.

```
BNFParser *BNFData :: CreateParser (BNFParser *bs_parent, logi-
                    cal list_symbols, char *trace_path )
```

bs_parent	Parent parser This is a pointer to the parent BNFParser object.
list_symbols	List symbols When this option is set to YES, the list of symbols is listed on console.
trace_path	Trace file location The location point to a file that contains all attempts for locating symbols in an expression. It records the symbol names and the symbol data.

GenerateSource - Generate source file for a parser

The function generates a source file that creates a parser for the BNF described by the BNFData tree. The BNFData tree can be constructed from a BNF definition file or string using the BNF meta parser (BNFDefinition).

```
logical BNFData :: GenerateSource (char *cpath )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
cpath	Complete path

The complete path is passed as 0-terminated string with a maximum length of 255 characters.

GetElement - Get Element

The function searches for an element in the BNFData tree. When terminating successfullly it retuns the BNFData object for the node foung, otherwise NULL.

i00 - Serch children by index

The function returnd the child of the BNFData node at position indx0. The first child has position 0.

```
BNFData *BNFData :: GetElement (int index0 )
```

i01 - Search node by name

The function searches for the nearest BNFData node using a symbol name. When no node with the symbol name passed was faond in the list of children the function searches recursively on all lower levels.

```
BNFData *BNFData :: GetElement (char *symbol )
```

GetSymbol - Looks for expression a given symbol (recursive)

The function returns the expression for the symbol when the sub-expression defined in the BNFData object corresponds to a symbol with the name passed in 'symbol' or not. The function searches recursively in subordinated symbols. Searchung stops, when a sub-expression has more than one elemenmt.

```
BNFData *BNFData :: GetSymbol (char *symbol )
```

GetValue - Provides the element value as string

The function returns the element value as string, when the element contains data. Otherwise the function returns NULL.

```
char *BNFData :: GetValue (char *string, int32 maxlen )
```

string	String area Pointer to the 0-terminated string area.
maxlen	Size of output buffer

Specifies the length of the buffer, the information should be stored into. The information is truncated if it is longer than the buffer.

GetValueLength - Get value length

The function returns the value length for the data area of the symbol.

```
int32 BNFData :: GetValueLength ( )
```

Return value

HasData - Has node data

The function returns, whether a BNFData node contains data or not.

```
logical BNFData :: HasData ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsA - Is expression a given symbol

The function returns, whether the sub-expression defined in the BNFData object exactly corresponds to a symbol with the name passed in 'symbol' or not.

```
logical BNFData :: IsA (char *symbol )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsValue - Has expressio a given value

The function returns, whether the sub-expression defined in the BNFData object has the value passed in 'string' or not.

```
logical BNFData :: IsValue (char *string )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
string	String area Pointer to the 0-terminated string area.

Position - Current position in string

```
char *BNFData :: Position ( )
```

Return value	Pointer to the 0-terminated string area.

Print - Print syntax tree

The function will print the syntax tree to console.

i0

```
logical BNFData :: Print (logical recursive )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
recursive	

i01

```
logical BNFData :: Print (char *path, logical recursive )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
path	
recursive	

i02

```
logical BNFData :: Print (FILE *fileptr, logical recursive, log-
                          ical erropt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
recursive	

BNFExpression - BNF expression

A BNF expression defines a BNF symbol. A BNF Symbol might be defined by more than one expression. A BNF expression may consist of one or more BNF elements with a defined order.

AddElement - Add element to expression.

```
int32 BNFExpression :: AddElement (BNFSymbol *bsymbol, logical
                 is_optional, int32 rep_count, logical
                 case_opt, logical sep_opt )
```

Return value	The validation code has the following values: 0 - matches symbol exactly 1 - matches symbol but contains additional spaces 2 - matches symbol including separating space, but contains more information 3 - matches symbol but contains further data without separating blank 4 - does not match symbol
bsymbol	BNF Symbol Pointer to a BNF symbol.
is_optional	Is value optional YES (or true) for this value indicates, that the item is optional.
rep_count	Repetition count Number of repetitions for the item.
case_opt	Case sensitive The option indicates case sensitive data in text (YES)
sep_opt	Separator option When this option is set to YES (true) a separator will be inserted between two items.

Analyze - Analyse expression

```
int32 BNFExpression :: Analyze (BNFExpression **bexpressions,
                 BNFData *bdata, int index0, logical case_opt )
```

Return value	The validation code has the following values:
	0 - matches symbol exactly
	1 - matches symbol but contains additional spaces
	2 - matches symbol including separating space, but contains more information
	3 - matches symbol but contains further data without separating blank
	4 - does not match symbol
bexpressions	
case_opt	Case sensitive
	The option indicates case sensitive data in text (YES)

NextExpressions -

```
int16 BNFExpression :: NextExpressions (BNFSymbol
                    *target_symbol, BNFExpression **bexpressions,
                    BNFData *bdata, int index0, logical case_opt )
```

Return value	
bexpressions	
case_opt	Case sensitive
	The option indicates case sensitive data in text (YES)

BNFParser - Parser for BNF strings

A string according to a given BNF syntax is based on a (top) BNF symbol. You may derive a specific BNF parsers for each type of BNF you want to support. The BNF is defined in the constructor for the BNF parser. Any number of spaces is allowed between symbols in a BNF but not required. Spaces are usually considered as separators between symbols.

BNF parsers can be referenced as symbols in other BNF symbols. This allows defining common BNF symbols e.g. for name and number (as BNFStandardSymbold). You may create a hierarchy consisting of a BNF tree by passing the parent (the more complex definition) to the referenced BNF or by constructing objects for referenced BNF parsers.

Analyze - Analyse string

The function analyses the string and creates a hierarchy of BNF data elements. Each BNF data element refers to the string, that describes the data element and to the symbol or token defined for the data element.

The function will parse the string according to the syntax defined in the BNF (top-aymbol) or according to a given symbol of the syntax.

i00 - Analyse expression

The function analyses the passed string as an expression of the syntax as being defined in the BNF (top-symbol).

```
BNFData *BNFParser :: Analyze (char *string, logical skip_sep )
```

string	String area Pointer to the 0-terminated string area.
skip_sep	Skip separators The option indicates that separators (blanks, tabs and new line characters) at the beginning of the expression should be ignored. Default: YES

i01 - Analyze symbol

The function analyses a substring of an expression according to the syntax defined for the symbol passed to the function.

```
BNFData *BNFParser :: Analyze (char *string, char *symbol, logi-
                 cal skip_sep )
```

string	String area Pointer to the 0-terminated string area.
skip_sep	Skip separators The option indicates that separators (blanks, tabs and new line characters) at the beginning of the expression should be ignored. Default: YES

AnalyzeFile - Analyse BNF definition provided in a file

The function analyses BNF definitions provided in the file passed to the function (path) and creates a hierarchy of BNF data elements. Each BNF data element refers to the string, that describes the data element and to the symbol or token defined for the data element.

i00

```
BNFData *BNFParser :: AnalyzeFile (char *path, logical skip_sep
                 )
```

path	
skip_sep	Skip separators The option indicates that separators (blanks, tabs and new line characters) at the beginning of the expression should be ignored. Default: YES

i01

```
BNFData *BNFParser :: AnalyzeFile (char *path, char *symbol,
                 logical skip_sep )
```

path	
skip_sep	Skip separators

The option indicates that separators (blanks, tabs and new line characters) at the beginning of the expression should be ignored.

Default: YES

BNFParser - Constructor

The constructor allows constructing a top-parser as well as a referenced parser. When constructing a referenced parser, you must pass the poiter to the top parser as 'bs_parent'.

```
                     BNFParser :: BNFParser (char *names,
                 BNFParser *bs_parent, logical skip_new_line,
                 logical term_opt, char *trace_path )
```

names	
bs_parent	Parent parser This is a pointer to the parent BNFParser object.
skip_new_line	New line as separator This option is set to true (YES), when new line characters (10,13) should be considered as separators (like blank and tab). In this case new line characters should not be used as BNF symbols. Default: YES
term_opt	Terminate symbol This option indicates, thet the symbol is a terminal symbol, i.e. subordinated noodes need not to be displayed in the syntax tree.
trace_path	Trace file location The location point to a file that contains all attempts for locating symbols in an expression. It records the symbol names and the symbol data.

GetLastError - Return last parser error

The function returns an error string for the last parser error. This information is overwritten when calling the Analyse function for the nect expression.

The function returns NULL, when no error has been set.

```
char *BNFParser :: GetLastError ( )
```

Return value	Pointer to the 0-terminated string area.

IsValidString - Checks, whether the symbol passed is valid

```
logical BNFParser :: IsValidString (char *symbol_name, char
                    *string, int32 string_len )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
string	String area Pointer to the 0-terminated string area.
string_len	String length The string length defines the maximum number of characters that can be stored in the string area without counting the terminating 0. Usually this value is 1 less that the allocated string area.

ListSymbols - List symbols for the parser

The function allows listing the symbols defined for a parser including symbols referenced from external parsers.

```
logical BNFParser :: ListSymbols ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ResetLastError - Reset last parser Error

The function resets the last parser error after it has been displayed or analysed.

```
void BNFParser :: ResetLastError ( )
```

~BNFParser - Destructor

```
BNFParser :: ~BNFParser ( )
```

BNFSymbol - BNF symbol

A BNF symbol is a variable in a BNF specification, which is defined by one or more BNF expressions. Symbols do have a neme and may consist of one or more expressions. Terminal symbols are tokens with a fixed value, which is stored in the name. Terminal symbols do not habe expressions.

BNFSymbol - Konstruktor

i0

```
                    BNFSymbol :: BNFSymbol (BNFParser
               *bparser, char *names, logical term_opt )
```

names

term_opt — Terminate symbol

This option indicates, thet the symbol is a terminal symbol, i.e. subordinated noodes need not to be displayed in the syntax tree.

i01

```
                    BNFSymbol :: BNFSymbol (
               )
```

ElementaryToken - Provide symbol for elementary token

The function provides the standard symbol for single characters (elementary tokens.

```
BNFSymbol *BNFSymbol :: ElementaryToken (uint8 ctoken, logical
               case_opt )
```

Return value	Pointer to a BNF symbol.
ctoken	Elementary token Elementary tokens are single characters fro 0-255. Symbols for elementary tokens are generated automatically.
case_opt	Case sensitive The option indicates case sensitive data in text (YES)

SetTerminal - Mark symbol as terminal symbol

Symbols marked as terminal symbols will not keep children symbols for analysing. Typically, keywords, which have a child for each character, are marked as terminal symbols.

```
void BNFSymbol :: SetTerminal ( )
```

~BNFSymbol - Destruktor

```
BNFSymbol :: ~BNFSymbol ( )
```

CTX_Base - General Context Class

The general context class is a base class for all database or GUI context classes. A context usually defines a data element or a data collection in its specific context, e.g. children in the context of a person or in a list in a GUI application. In a context the behaviour of object becomes more specific, which can be expressed in a context class. Context classes have to be defined in a specified form according to the type of context to be implemented.

Contexts in an application form a hierarchy, i.e. each context object has either a parent (upper) context or is a top context (e.g. database or project context).

Each context has two status properties which reflect the current state of the context. Since context classes area created and deleted by the system the current state of a context class (as opened or closed) is not always clear for the application programmer. The process state (-> CTX_ProcessState) describes the the current state in the processing. The display state (-> CTX_DisplayState) describes the visibility of the context. For GUI context classes this is the way the associated GUI element is presented at the moment on the user interface. State properties are maintained by the system but can be retrieved by the application.

Moreover, the context class provides three user states that can be updated and retrieved by the application.

Context classes signal several events that are relevent for the specific context. Thus, context classes are typically used for handling system events as delete or inserted for database instances or lose/get focus for GUI contexts. The enumeration of events supported by a context class is defined in the cpecific context class implementation.

The context class supports the action interface, which enables functions in context classes calling actions defined in a reasource database or created internally.

CTX_Base - Konstruktor

The constructor for a general context class should never be called explicitly, but only by a specialized class.

```
CTX_Base :: CTX_Base (
)
```

CheckPermission - Check Permission

The function checks whether the user/application has permission for running the action passed to the function. The function returns true (YES) when the application has permissions for calling the action.

Permissions are defined in the project or database context and must be initialized when permission check is to be supported.

```
logical CTX_Base :: CheckPermission (UCA_Action *actptr )
```

Return value	When this value is true the function will continue, otherwise the processing terminates.
actptr	Action pointer The pointer refers to an internal action that has been implicitely defined or created from a resource.

Close - Close Handler

The function is called when the context is going to be closed. The handler can be overloaded in the specific context to perform necessary actions before deleting the context. In this phase all resources of the context are still accessible.

```
logical CTX_Base :: Close ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

CreateAction - Create action from action resource

The function allows creating simple actions from a resource action definition.

```
UCA_Action *CTX_Base :: CreateAction (SimpleAction *action )
```

Return value	The pointer refers to an internal action that has been implicitely defined or created from a resource.

action	Simple Action The simple action defines the context action and the action type. Some action types require more detailled action definitions that will be retrieved in the dictionary. In this case the dictionary must contain an appropriate action definition.

CreateCAction - Create complex action

The function creates a complex action including pre and post handler for the action.

```
UCA_CAction *CTX_Base :: CreateCAction (SimpleAction
                 *prehandler, SimpleAction *action, SimpleAc-
                 tion *postandler )
```

Return value	A complex action defines an internal action including pre- and post-handler.
prehandler	Pre-handler The pre-handler is passed as simple action containing the action name and the action type.
action	Simple Action The simple action defines the context action and the action type. Some action types require more detailled action definitions that will be retrieved in the dictionary. In this case the dictionary must contain an appropriate action definition.
postandler	Post-handler The post-handler is passed as simple action containing the action name and the action type.

DataState - Get data state

The function returns the current data state for the context.

```
CTX_DataStates CTX_Base :: DataState ( )
```

Return value

DeleteData - Delete data handler

The delete data handler is called before deleting a data element. The handler can be overloaded in specialized context class implementations.

The delete handler can deny the data deletion by returning YES.

```
logical CTX_Base :: DeleteData ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DisplayDecision - Display decision

The function displays a decision according to the environment (message box for windows applications and command line decision for console applications). The message text is defined in the message number that refers to a defined message in the resource database. When passing no message number or 0 the message is constructed from the parameters passed to the function.

The possible decisions can be defined by means of predefined reply combinations. One of the replys can be defined as default reply. The selected reply is returned to the upplication that waits until the user has been replied.

```
ReplyTypes CTX_Base :: DisplayDecision (int16 msgnum, ReplyCombi
                  buttons, ReplyTypes def_dec, char *parm1, char
                  *parm2, char *parm3 )
```

Return value	The decision is returned as reply type that is associated with the given reply.
msgnum	
buttons	Decision combinations The decision allows different combinations to be displayed in the decision, which ere described as "Reply Combinations".
def_dec	Default reply One of the replys displayed in the decision can be set as default reply.
parm1	

parm2

parm3

DisplayMessage - Display message

The function allows displaying a message for a signaled or passed message code. Depending on the environment the error is written to the console (console applications) or displayed in a message box (GUI application).

```
void CTX_Base :: DisplayMessage (int16 w_msgnum, char *parm1,
                                 char *parm2, char *parm3 )
```

w_msgnum	Message code The message code passed must be a defined error code in the reesource database.
parm1	
parm2	
parm3	

DisplayState - Get display state

The function returns the current display state for the context.

```
CTX_DisplayState CTX_Base :: DisplayState ( )
```

Return value

ExecuteAction - Execute action

The function allows executing an action with the action name and type passed to the function or an event. The function returns whether the action could be exuted faormally. The action result can be retrieved with the function GetActionResult().

FNCACTE_ - Execute action by name

The function calls an action by passing the action type and the action name. In some cases actions require an action resource that provides more information (Wiondow or document action). When calling more complex actions the action must be defined in an action resource, which is available in the resource database.

```
logical CTX_Base :: ExecuteAction (char *acnames, ActionType ac-
                 type )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
actype	

i1 - Execute event action

The function calls passes an event which calls the associated event handler for the context. Since internal events define typical events not all events are supported for all context types. Hence, events passed to the function must be checked for the specific context type.

```
logical CTX_Base :: ExecuteAction (InternalEvent eventid )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
eventid	Event identifier The event identifier is an internal number that is defined for typical events.

ExecuteFunction - Execute function

The function calls a context function that has been defined as action (function action).

FNCEXE - Context action function interface

This function must be overloaded in the specific context class, as soon as the context class defined action functions which can be triggered by events. Usually, this function is generated by the development environment. The example below shows a typical implementation.

```
logical CTX_Base :: ExecuteFunction (char *fname, logical
                 chk_opt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
fname	

i1 - OQL function interface

This function interface must be implemented to enable context functions for being called in OQL functions.

```
logical CTX_Base :: ExecuteFunction (char *fname, int16 parmcnt,
                    PropertyHandle **parmlist, PropertyHandle
                    &retfld )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
fname	
parmcnt	Number of parameters The number of parameters must correspond to the numbers of parameters in the subsequent parameter list.
parmlist	
retfld	Return value The return value area is passed as property handle.

ExecuteProgram - Execute program

The function calls a windows or console program or batch file as passed in the program path. The control is returned to the application as soon as the program has been started.

```
logical CTX_Base :: ExecuteProgram (char *prgnams, char
                    *pgmparm1, char *pgmparm2, char *pgmparm3,
                    char *pgmparm4, char *pgmparm5 )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prgnams	Program path

The complete path for the program to be called is passed as 0-terminated string.

pgmparm1

pgmparm2

pgmparm3

pgmparm4

pgmparm5

FillData - Fill data handler

The fill data handler is called when a data instance or GUI element has been filled with data. The handler can be overloaded in specialized context class implementations.

```
logical CTX_Base :: FillData ( )
```

Return value — The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

GetActionResult - Get last action result

The last action result is stored in an internal field which allows context functions or complex actions reacting on the result returned from the action. Some return values are interpreted when executing complex actions as follows:

YES: action terminated with error

NO: Action terminated normally

AUTO: action not executed because pre-handler terminated with error.

```
int32 CTX_Base :: GetActionResult ( )
```

Return value

GetContextType - Get context type

The function returns the context type for the current context class as e.g. CTX_Property or CTX_Windoww.

```
CTX_Types CTX_Base :: GetContextType ( )
```

Return value	The context type for the context class describes the application resource reflected by the context. Default: CTXT_undefined

GetDecision - Get decision

The function creates an message from the mesage number and the passed variables and generates a decision that is displayed on the console for console applications or in a message box for GUI applications. The function retruns true (YES), when the response was 's' or 'S' (for si), 'o' or 'O' (for oui), 'y' or 'Y' (for yes) or 'j' or 'J' (for ja) or when the YES/OK button has been pressed and false (NO) otherwise.

```
logical CTX_Base :: GetDecision (int16 msgnum, char *parm1, char
                    *parm2, char *parm3 )
```

Return value

msgnum

parm1

parm2

parm3

GetMessageString - Get message string

The function creates a message string for the passed message number. Parameters for replacing message variables defined in the message resource can be passed to the function.

```
char *CTX_Base :: GetMessageString (int16 msgnum, char *parm1,
                    char *parm2, char *parm3 )
```

msgnum

parm1

parm2

parm3

GetPropertyHandle - Get Property handle

The function returns the property handle associated with the data for the context. The function cannot be called for project, application, database or database object context, since the data associated with those context cannot be described by means of a property handle.

When a property name is passed to the function the subordinated property handle for the context property handle is returned. The name passed to the function must be a valid property name in the structure/class defined for the context property.

```
PropertyHandle *CTX_Base :: GetPropertyHandle (char *fldname_w )
```

Return value

fldname_w	Property name or path The property name is passed as 0-terminated string. Default: ""

GetResourceName - Get resource name

The function returns the context specific resource name.

```
char *CTX_Base :: GetResourceName ( )
```

Return value

HighContext - Get parent context

The function returns the next upper context with the context type and/or resource name passed to the function.

FNCHIGHG - Get upper context with defined type

The function looks for the next higher context with the context type passed to the function. When no context type is passed (CTX_undefined) the function returns the next higher context.

```
CTX_Base *CTX_Base :: HighContext (CTX_Types ctxtype )
```

Return value

ctxtype	Context type

The context type for the context class describes the application resource reflected by the context.

Default: CTXT_undefined

i1 - Get upper resource context

The function searches for the next higher context with a given resource name and (when passing a defined context type) the context type passed to the function.

```
CTX_Base *CTX_Base :: HighContext (char *resname, CTX_Types
                    ctxtype )
```

Return value	
resname	Resource name The resource name is passed as 0-terminated string with a maximum length of 40 characters.
ctxtype	Context type The context type for the context class describes the application resource reflected by the context. Default: CTXT_undefined

InitData - Init data handler

The init data handler is called when a data instance or GUI element has been initialized. The handler can be overloaded in specialized context class implementations.

```
logical CTX_Base :: InitData ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

InsertData - Insert data handler

The insert data handler is called before inserting a data element. The handler can be overloaded in specialized context class implementations.

The insert data handler can deny the data insertion by returning YES.

```
logical CTX_Base :: InsertData ( )
```

Return value

The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

IsEdit - Can data be updated

The function checks whether data can be updated in the given context.

```
logical CTX_Base :: IsEdit ( )
```

Return value

The function returns YES when the question was answered positivly. Otherwise it returns NO.

NextData - Next data handler

The next data handler is called when the next data instance or GUI element has been located. The handler can return an error (YES) to force the system providing the next instance. Thus, the next handler allows implementing data filters in a given context.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_Base :: NextData ( )
```

Return value

The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Open - Open context

The function is called when the context has been opened. The handler can be overloaded in the specific context to perform necessary actions after opening the context. In this phase all resources of the context are already accessible.

```
logical CTX_Base :: Open ( )
```

Return value

The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Parm - Provides parameter

When calling context actions parameter cannot be passed directly. The application must use the SetParm function to pass parameters to the context action. Parameters that have been set with the SetParm function can be retrieved with the Parm() function as keyword or position parameters.

After retrieving a parameter value it can be used until the next parameter is retrieved. Copy the parameter value when it is still needed. Do not refer by pointer to several parameters at the same time.

i00

```
char *CTX_Base :: Parm (int32 parm_no )
```

Return value	The parameter string is passed as 0-terminated string and contains the parameters according to the conventions of the action called.

i01

```
char *CTX_Base :: Parm (char *parm_key )
```

Return value	The parameter string is passed as 0-terminated string and contains the parameters according to the conventions of the action called.
parm_key	Parameter keyword The keyword that is searched in the parameter list. The keyword is passed as 0-terminated string. Parameter keywords are not case sensitive.

PreviousData - Previous data handler

The previous data handler is called when the previous data instance or GUI element has been located. The handler can return an error (YES) to force the system providing anoter previous instance. Thus, the previous handler allows implementing data filters in a given context.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_Base :: PreviousData ( )
```

Return value — The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ProcessState - Get process state

The function returns the current process state for the context.

```
CTX_ProcessState CTX_Base :: ProcessState ( )
```

Return value

ResetData - Reset data handler

The reset data handler is called when a data instance or GUI element has been reset. The handler can be overloaded in specialized context class implementations.

```
logical CTX_Base :: ResetData ( )
```

Return value — The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

SaveData - Save data handler

The save data handler is called when a data instance or GUI element has been saved. The handler can be overloaded in specialized context class implementations.

```
logical CTX_Base :: SaveData ( )
```

Return value — The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

SelectData - Select data handler

The select data handler is called when a data instance or GUI element has been selected. The handler can be overloaded in specialized context class implementations.

```
logical CTX_Base :: SelectData ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

SetActionResult - Set action result

The function must be called from context handlers or action functions to set the return value for the action in the context. This retrun value can be retrieved by other context functions until the nect action call.

Valid values for the return code are

0 or NO - action terminated normally

1 or YES - action terminated with error

```
void CTX_Base :: SetActionResult (int32 rc )
```

rc

SetDataState - Set data state

Usually the data state is maintained by the system. Context functions, however, may set data states to inform the system e.g. that data has become invalid.

The function returns the previuos data state.

```
CTX_DataStates CTX_Base :: SetDataState (CTX_DataStates datstate
                           )
```

Return value

datstate

SetDisplayState - Set display state

Usually the display state is maintained by the system. Context functions, however, may set display states e.g. to request data or GUI elements to be hidden further on.

The function returns the previuos display state.

```
CTX_DisplayState CTX_Base :: SetDisplayState (CTX_DisplayState
                            dspstate )
```

Return value	The data state is set to DSP_disabled, when the context is set to read only.
dspstate	Data state

The data state is set to DSP_disabled, when the context is set to read only.

SetParm - Set Parameters for context action

When calling context actions parameter cannot be passed directly. The application must use the SetParm() function to pass parameters to the context action. This function is usually called internally from the ExecuteAction function for different handles.

```
logical CTX_Base :: SetParm (char *parm1, char *parm2, char
                 *parm3, char *parm4, char *parm5 )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
parm1	
parm2	
parm3	
parm4	
parm5	

SetUserState1 - Set first user state

The function set the new value for the user state and returns the previous user state.

```
int16 CTX_Base :: SetUserState1 (int16 userstat )
```

Return value	The user state is true (YES) or false (NO).
userstat	User state The user state is true (YES) or false (NO).

SetUserState2 - Set second user state

The function set the new value for the user state and returns the previous user state.

```
int16 CTX_Base :: SetUserState2 (int16 userstat )
```

Return value	The user state is true (YES) or false (NO).
userstat	User state The user state is true (YES) or false (NO).

SetUserState3 - Set third user state

The function set the new value for the user state and returns the previous user state.

```
int16 CTX_Base :: SetUserState3 (int16 userstat )
```

Return value	The user state is true (YES) or false (NO).
userstat	User state The user state is true (YES) or false (NO).

SetupParm - Setup parameter list options

The function allows settting up the parameter list by defining separator and parameter list type. The default values are ',' as seperator and no keyword parms. When another type of parameter list should be used for passing parameters to context actions of the given context it is suggested to setup the parameter list when opening the context (DBOpened).

```
logical CTX_Base :: SetupParm (char separator, logical key_parms
                   )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
key_parms	Parmlist with key parameter format This option is set to YES (true) when the parameter list contains key word parameters. If no (position parameters) the option is set to NO (false).

StoreData - Store data handler

The store data handler is called when a data instance or GUI element has been saved. The handler can be overloaded in specialized context class implementations.

```
logical CTX_Base :: StoreData ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

UserState1 - Get first user state

The function returns the current user state.

```
int16 CTX_Base :: UserState1 ( )
```

Return value

UserState2 - Get second user state

The function returns the current user state.

```
int16 CTX_Base :: UserState2 ( )
```

Return value

UserState3 - Get third user state

The function returns the current user state.

```
int16 CTX_Base :: UserState3 ( )
```

Return value

CTX_DBBase - Base class for data base contexts

The base class for database contexts provides some basic functionality for data base context classes. In particular the class provides most of the default event handlers that can be overloaded in specific context classes. Overloaded handlers need not to call the default handlers since nothing is done in the default handlers. Handlers for database events are usually called within internal transactions. Thus, all modifications made by the event handler are reset when the transaction fails.

For a number of database operations Not-events are generated that are called in case of an error. An error could be a database (consistency) error but the process event (before event) could have denied the operation as well.

CheckPermission - Check permission

The function checks whether the user/application has permission for running the action passed to the function. The function returns true (YES) when the application has permissions for calling the action.

Permissions are defined in the project or database context and must be initialized when permission check is to be supported.

```
logical CTX_DBBase :: CheckPermission (UCA_Action *actptr )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
actptr	Action pointer The pointer refers to an internal action that has been implicitely defined or created from a resource.

DBBeforeRead - Before read event handler

The before read event handler is called before reading an instance (DBP_Read event).

At the time, when the handler is called, the instance is already selected in the property handle. Key data for the sort key (when defined) is available and can be copied to the instance area using the SetKey() structure context function or can be provided by using the GetKey() context function.

The handler can be used to optimize read access by returning YES or marking an instance as 'hidden' (HideInstance()), when an instance with the given key should not be provided.

The handler can be overloaded in specialized structure context class (CTX_Structure) implementations.

```
logical CTX_DBBase :: DBBeforeRead ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBClose - Close event

The function reacts on the DBO_Close event, i.e. it is called when the context is going to be closed. In this phase all resources of the context are still accessible.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBClose ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBCreate - Before create handler

The before create handler is called before creating a new data instance (DBP_Create event).

The before create handler can deny creating the data instance by returning YES.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBCreate ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBCreated - After create handler

The after create handler is called when a data instance has been cretaed. (DBO_Created event). The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBCreated ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBDelete - Before delete handler

The before delete handler is called before deleting a data instance (DBP_Delete event).

The before delete handler can deny deleting the data instance by returning YES.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBDelete ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBDeleted - After delete handler

The after delete handler is called when a data instance has been deleted. (DBO_Deleted event). The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBDeleted ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBInitialize - Initialize handler

The initialize handler is called when a data instance has been initializes (DBO_Initialized event). When the handler is called the instance is not yet selected in the property handle. Hence, only attributes can be accessed in the instance in this phase. References and relationships are not accessable.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBInitialize ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBInsert - Before insert handler

The before insert handler is called before insering a data instance in a collection (DBP_Insert event). When the handler is called the instance to be inserted in the collection is not yet selected in the property handle. Hence, only attributes can be accessed in the instance in this phase. References and relationships are not accessable.

The before insert handler can deny creating the data instance by returning YES.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBInsert ( )
```

Return value

The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBInserted - After inserted handler

The after inserted handler is called when a data instance has been inserted in a collection (DBO_Inserted event). In contrast to create insert means only that an instance has been added to a collection but it need not to be a newly created instance.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBInserted ( )
```

Return value

The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBModify - Modify handler

The modify handler is called when a modification is signaled for the instance (DBP_Modify event). This may happen before or after performing the modification. new data instance.

The before create handler can deny creating the data instance by returning YES.

The handler can be overloaded in specialized context class implementations. When handling the modify event no values should be assigned to the instance of the property handle since this will cause another modification event and thus, a recursive call of the event handler. You can prevent recursion by setting a user state and checking it always when entering the modify event handler. The state should be reset at least in the stored handler to handle new modify events.

```
logical CTX_DBBase :: DBModify ( )
```

Return value

The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBNotCreated - Not created handler

The not created handler is called when a data instance could not be cretaed because of an error (DBO_NotCreated event). The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBNotCreated ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBNotDeleted - Not deleted handler

The not deleted handler is called when a data instance could not be deleted because of an error (DBO_NotDeleted event). The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBNotDeleted ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBNotInserted - Not inserted handler

The not inserted handler is called when a data instance could not be inserted because of an error (DBO_NotInserted event). The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBNotInserted ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBNotOpened - Not opened handler

The not opened handler is called when the context could not be oened because of an error (DBO_NotOpened event). The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBNotOpened ( )
```

Return value — The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBNotRemoved - Not removed handler

The not removed handler is called when a data instance could not be removed from a collection because of an error (DBO_NotRemoved event). The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBNotRemoved ( )
```

Return value — The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBOpen - Before open handler

The before open handler is called when opening the context (DBP_Open event).

The before delete handler can deny opening the context by returning YES. In this case the cantext has the process state PRC_NotOpened.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBOpen ( )
```

Return value — The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBOpened - After Open handler

The function is called when the context has been opened (DBOpened event). The handler can be overloaded in the specific context to perform necessary actions after opening the context. In this phase all resources of the context are already accessible.

```
logical CTX_DBBase :: DBOpened ( )
```

Return value — The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBRead - After read event

The after delete handler is called when a data instance has been deleted. (DBO_Read event). This handler is typically used to initialize transient attributes and references in the instance or to select propert settings for generic attributes.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBRead ( )
```

Return value — The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBRefresh - Refresh handler

The refresh handler is signaled by the application when submitting a refresh request to a property handle (DBO_Refresh event). This handler is typically used to initialize transient attributes and references for the property handle or to re-calculate derived values.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBRefresh ( )
```

Return value — The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBRemove - Before remove handler

The before remove handler is called before removing a data instance from a collection (DBP_Remove event).

The before insert handler can deny creating the data instance by returning YES.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBRemove ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBRemoved - After remove handler

The after remove handler is called when a data instance has been removed from a collection (DBO_Removed event). In contrast to delete remove means only that an instance has been removed from a collection but not necessarily deleted as instance. When the handler is called the instance removed from the collection is not anymore selected in the property handle and thus, not accessible.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBRemoved ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBStore - Before stor handler

The before store handler is called before storing data to the transaction or database (DBP_Store event). The event handler is typically used to perform application consistency checks for the instance. Since in this phase all indices have already been updated and consistency checks have been finished key components must not be updated. When changing sub-ordinated instances in this handler the modifications should be saved explicitely. Otherwise, they are not stored in the same transaction and may cause problems when the transaction fails.

The before store handler can deny creating the data instance by returning YES.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBStore ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBStored - After strore handler

The after store handler is called when a data instance has been stored (DBO_Stored event). The handler can be overloaded in specialized context class implementations.

```
logical CTX_DBBase :: DBStored ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ExecuteAction - Execute action

The function allows executing an action with the action name and type passed to the function or an event. The function returns whether the action could be exuted faormally. The action result can be retrieved with the function GetActionResult().

```
logical CTX_DBBase :: ExecuteAction (DB_Event intevent )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
intevent	Event identifier The event identifier is an internal number that is defined for typical events.

GetPropertyHandle - Get Property handle

The function returns the property handle associated with the data for the context. The function cannot be called for database or database object context, since the data associated with those context cannot be described by means of a property handle.

When a property name is passed to the function the subordinated property handle for the context property handle is returned. The name passed to the function must be a valid property name in the structure/class defined for the context property.

```
PropertyHandle *CTX_DBBase :: GetPropertyHandle (char *fldname_w
                 )
```

Return value	Is a pointer to an (usually) opened property handle.
fldname_w	Property name or path The property name is passed as 0-terminated string. Default: ""

GetSysDict - Get system dictionary

The function returns the system dictionary for the database opened. The system dictionary contains the model definitions for the metadata.

```
DictionaryHandle &CTX_DBBase :: GetSysDict ( )
```

Return value	This is a reference to an opened dictionary handle.

HighDBContext - Get next higher database context

The function returns the next upper context with the context type and/or resource name passed to the function.

i0

```
CTX_DBBase *CTX_DBBase :: HighDBContext (CTX_Types ctxtype )
```

ctxtype	Context type The context type for the context class describes the application resource reflected by the context. Default: CTXT_undefined

i1

```
CTX_DBBase *CTX_DBBase :: HighDBContext (char *resname,
                                  CTX_Types ctxtype )
```

resname	Resource name The resource name is passed as 0-terminated string with a maximum length of 40 characters.
ctxtype	Context type The context type for the context class describes the application resource reflected by the context. Default: CTXT_undefined

SetTransactionError - Set trasaction error

The function marks a transaction as errounus. This leads to a rollback of the transaction when the transaction is finished. The function can be called in post handlers (as inserted or deleted) to undo the performed operation.

```
void CTX_DBBase :: SetTransactionError ( )
```

CTX_DataBase - Database Context

The database context allows defining functionality that is executed when opening or closing a database. The database context does not have a parent context.

The default database context can be overloaded by a application specific database context class.

CTX_DataBase - Konstructor

```
CTX_DataBase :: CTX_DataBase (
)
```

GetDBHandle - Det database handle

The function returns a database handle for the database.

```
DatabaseHandle &CTX_DataBase :: GetDBHandle ( )
```

Return value

~CTX_DataBase - Destructor

The function destroys the database context. The function must be overloaded in an application specific implementation of the database context.

```
CTX_DataBase :: ~CTX_DataBase ( )
```

CTX_Object - Database Object Context

The Database Object context allows defining functionali-ty that is executed when opening or closing a Database Object. The parent context for an object context is an object con-text (if the Database Object is not the root Database Object) or the database context (for the root Database Object).

The default database object context can be overloaded by a application specific database context class.

~CTX_Object - Destructor

The function destroys the database object context. The function must be overloaded in an application specific implementation of the database object context.

```
CTX_Object :: ~CTX_Object ( )
```

CTX_Property - Property contexts

Property contexts are created for extents, references, attributes, relationships and base structures. The property context defines refers to the property instance as well as to the property definition. Moreover, it allows determining the active context hierarchy for the property, i.e. the parent structure/Database Object, the property the parent structure is accessed from, the parent parent structure etc. Thus, the property context defines the context in which the property instance has been provided.

The parent context for a property context is a structure context (when the property is part of an object instance) or a Database Object context (when the property is an extent.

The property context allows handling read and update events, validity checks and insert and remove events.

The default property context can be overloaded by a application specific property context classes.

DBRefresh - Refresh handler

The refresh handler is signaled by the application when submitting a refresh request to a property handle (DBO_Refresh event). This handler is typically used to initialize transient attributes and references for the property handle or to re-calculate derived values.

The handler can be overloaded in specialized context class implementations.

```
logical CTX_Property :: DBRefresh ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

GetInstContext - Get Instance context

The function returns the structure context for the instance that owns the current property (high structure). This is not identical with the structure that owns the property. The instance owning the property is the instance that is stored in the database. Hence, the function goes up in the context hirarchy until it finds the context that referst to the instance stored in the database.

```
CTX_Structure *CTX_Property :: GetInstContext ( )
```

Return value	Structure context for a property handle.

GetPropContext - Get Property context

The function returns the property context for the property passed as name or property path. The property is searched in the structure that owns the current property.

```
CTX_Property *CTX_Property :: GetPropContext (char *w_fldnames )
```

Return value	This is the default property context or a user-defined context class instance for the property.
w_fldnames	Property path or name The property name is passed as 0-terminated string. It may contain a property path that consists of a sequence of property names. NULL is passed if no property name is available.

GetPropertyHandle - Get Property Handle

The function returns the property handle associated with the data for the context. The function cannot be called for database or database object contexts, since the data associated with those contexts cannot be described by means of a property handle.

When a property name is passed to the function the subordinated property handle for the context property handle is returned. The name passed to the function must be a valid property name in the structure/class defined for the context property.

```
PropertyHandle *CTX_Property :: GetPropertyHandle (char
                 *fldname_w )
```

Return value	Is a pointer to an (usually) opened property handle.
fldname_w	Property name or path

The property name is passed as 0-terminated string.

Default: ""

GetResourceName - Get resource name

The function returns the property name as context specific resource name.

```
char *CTX_Property :: GetResourceName ( )
```

Return value	The resource name is passed as 0-terminated string with a maximum length of 40 characters.

GetStructContext - Get structure context

The function returns the structure context for the currently selected instance in the property handle. For weak-typed properties the context may change with the selection from instance to instance. When no instance is selected in the property handle associated with the context the function returns the instance context for the default instance.

```
CTX_Structure *CTX_Property :: GetStructContext (char
                     *w_strnames )
```

Return value	Structure context for a property handle.
w_strnames	

IsEdit - Can data be updated

The function checks whether data can be updated in the property handle.

```
logical CTX_Property :: IsEdit ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsReadOnly - Is read-only enabled

The function returns whether the read only option has been set in the instance.

```
logical CTX_Property :: IsReadOnly ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

SetReadOnly - Set read only

The function allows setting the instance selected in the property handle to read-only. This will prevent the data in the property handle from being updated. The indication is reset automatically, when reading the next instance. The function sets the read only optin for all subordinated property handles.

```
logical CTX_Property :: SetReadOnly (logical readonly )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
readonly	

SetResult - Set value for action result

Usually it is not possible to pass an result from a context function. You can, however, return a string list which can be retrieved by the application using the function GetActionResult(). The result is also passed from the server to the client when the action is executed on a server.

```
void CTX_Property :: SetResult (char *result_string )
```

result_string	Result string

The result string can be a list of strings where strings are usually separated by x01 characters. If there is only one string returned the string is 0-terminated. Multiple strings are terminated with 0 after the last string in the list, which should be terminated with x01 as well.

SetTransactionError - Set Transaction Error

The function marks a transaction as errounus. This leads to a rollback of the transaction when the transaction is finished. The function can be called in post handlers (as inserted or deleted) to undo the performed operation.

```
void CTX_Property :: SetTransactionError ( )
```

CTX_Structure - Structure Context

A structure context is created for each structure type. It defines the connection between the instance and the instance description. Moreover, it allows determining the active con-text hierarchy for the structure instance, i.e. the parent property/extent, the structure the parent property is defined in, the parent parent property etc. Thus, the structure context defines the context in which the object instance has been provided.

The parent context for a structure context is always a property context. This can be the property context for an extent or for another property within a structure instance.

The structure context allows handling read and updating events as well as creating or deleting events.

BuildObjDescription - Create an object description

The function provides a html or simple text description for the object that is constructed according to a defined template that describes the elements to be included into the object.

```
logical CTX_Structure :: BuildObjDescription (PropertyHandle
                   &templ_pi, logical html )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
templ_pi	Template property handle The property handle refers to the template that is used for buildung the textual presentation of the object.
html	HTML option Indicates whether the description (textual object presentatin) is to be provided in HTML format (YES) or not (NO).

CTX_Structure - Structure context constructor

```
                 CTX_Structure :: CTX_Structure (
             )
```

CopyTo - Duplicate instance

The function duplicates an existing instance. Since specific copy rules must be implemented in several cases an overloaded specific action can be defined in derived context classes. The function is not a virtual function and must be implemented as an action that can be called via the executeFunction function

```
logical CTX_Structure :: CopyTo ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

GetContextType - Get Context type

The function returns the specific context type as CTX_Structure.

```
CTX_Types CTX_Structure :: GetContextType ( )
```

Return value	The context type for the context class describes the application resource reflected by the context. Default: CTXT_undefined

GetInstContext - Get Instance context

The function returns the structure context for the instance that owns the current property (high structure). This is not identical with the structure that owns the property. The instance owning the property is the instance that is stored in the database. Hence, the function goes up in the context hirarchy until it finds the context that referst to the instance stored in the database.

```
CTX_Structure *CTX_Structure :: GetInstContext ( )
```

Return value	Structure context for a property handle.

GetInstance - Get instance

The function returns the instance for the context.

GetKey - Get Key value

This function can be called when an instance is selected in the property handle (PropertyHandle::IsSelected()) or in the DBBeforeRead() event handler. The function returns the key for the selcted instance in the internal key structure, when the collection is ordered or an empty key instance, when the collection is not ordered or no instance is selected in the collection.

```
char *CTX_Structure :: GetKey ( )
```

Return value	The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the ({.r pib.StringToKey}()) function. Regardles on the type key values are passed as (char *) areas.

GetOldField - Provide property handle for old instance

The function returns a froperty handle for the old instance (as currently stored in the database, -> GetOldInstance()).

```
PropertyHandle CTX_Structure :: GetOldField ( )
```

Return value	The property handle refers to data and metadata of the selected property.

GetOldInstance - Get old instance

The function returns the "old instance" state as it is stored still in the database. This allows comparing old and new values within the DBModify() or DBStore() handler.

```
char *CTX_Structure :: GetOldInstance ( )
```

Return value

GetPropContext - Get Property context

The function returns the property context for the property passed as name or property path. The property is searched in the structure associated with the context.

```
CTX_Property *CTX_Structure :: GetPropContext (char *w_fldnames
                                )
```

Return value	This is the default property context or a user-defined context class instance for the property.

w_fldnames	Property path or name The property name is passed as 0-terminated string. It may contain a property path that consists of a sequence of property names. NULL is passed if no property name is available.

GetPropertyHandle - Get Property handle

The function returns the property handle associated for the structure. When a property name or path is passed to the function the property handle is searched in the structure. When no property name or path is passed the property handle the structure belongs to is returned.

```
PropertyHandle *CTX_Structure :: GetPropertyHandle (char
                 *fldname_w )
```

Return value	Is a pointer to an (usually) opened property handle.
fldname_w	Property name or path The property name is passed as 0-terminated string. Default: ""

GetReadOnly - Is context set to read-only?

The function returns whether the context has been set to read-only (-> SetReadOnly()).

```
CTX_DisplayState CTX_Structure :: GetReadOnly ( )
```

Return value	The data state is set to DSP_disabled, when the context is set to read only.

GetRefContext - Get referenced context

Some functions as Copy are setting a reference context. The reference context can be set also explicitly using the SetRefContext() function. The function returns the reference context when it is set or NULL otherwise.

```
CTX_Structure *CTX_Structure :: GetRefContext ( )
```

Return value	Structure context for a property handle.

GetResourceName - Get resource name

The function returns the structure name as context specific resource name.

```
char *CTX_Structure :: GetResourceName ( )
```

Return value	The resource name is passed as 0-terminated string with a maximum length of 40 characters.

GetSourceField - Get source field

Some functions as Copy are setting the property handle for the source during the action that refers to a source property handle. The function returns the source property handle when it is set or an empty property handle otherwise.

```
PropertyHandle CTX_Structure :: GetSourceField ( )
```

Return value	The property handle refers to data and metadata of the selected property.

HideInstance - Hide instance

The function can be used in the structure context to exclude an instance from being selected in any property handle. Hidden instances will return NO when trying to lacate such an instance using the Get() or another function to locate the instance. Position() (or the ++ or -- operator) will skip hidden instances.

The state is typically set in the DBRead() handler. When resetting or changing the selection in the property handle the state is automatically reset. You may, however, reset the state explicitely using the ShowInstance() context function.

The function returns the 'hidden' state as it was set before calling the function.

```
logical CTX_Structure :: HideInstance ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsEdit - Can data be updated

The function checks whether data can be updated in the instance selected for the property handle.

```
logical CTX_Structure :: IsEdit ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsHidden - Is instance hidden

Hidden instances are not selected when attempting to read them. The state can be set using the structure context function HideInstance() in a derived structure context. The function returns YES, when the instance is hidden and NO otherwise.

```
logical CTX_Structure :: IsHidden ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsReadOnly - Has instance been set to read-only

The function returns whether the property handle has been set to read-only for instances (-> SetReadOnly()).

```
logical CTX_Structure :: IsReadOnly ( )
```

Return value

SetKey - Set key in instance area

This function can be called when an instance is selected in the property handle (PropertyHandle::IsSelected()) or in the DBBeforeRead() event handler. The function moves the key to the property instance area for the key components. The function returns an error (YES), when the collection is not ordered or when no instance is selected in the collection.

```
logical CTX_Structure :: SetKey ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

SetReadOnly - Set instance to read-only

The function allows protecting an instance from being updated or remove the write protection. The function can be called from within any structure context function or in relation with a structure context. The function will protect the current property handle and all subsequent handles, i.e. it works recursively. When resetting the read-only state the function will not reset subsequent property handles that have been set explicitly to read-only. Setting the read-only state for a property handle has the consequence that not only all subordinated instaces are locked for writing but all subordinated collections as well, i.e. that the application cannot add or delete instances from subordinated property handles.

The read-only state is set for the property handle, i.e. after setting the read only state all instances selected for the property handle are read-only until the state is reset by another context function call.

The function returns the current state for the property handle.

When changing the instance state for a property handle this will affect the write permission only, when being set before selecting an instance in the Property handle. To activate the state for the instance currently selected the instance can be re-selected (e.g. using {.r PropertyHandle.Reset}()). Resetting the read-only state will not affect instances that are write protected for other reasons and instances selected in other property handles, which have been set explicitely to 'read only' by the application.

```
logical CTX_Structure :: SetReadOnly (logical readonly )
```

Return value

readonly

SetRefContext - Set reference context

The function allows setting a reference context as link in another context e.g. to link structure contexts in a copy process. Only one context can be set as reference context. Calling the function several times will overwrite the reference context each time the function is called. When passing NULL as reference context the reference context will be reset.

```
logical CTX_Structure :: SetRefContext (CTX_Structure *strctx )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
strctx	Structure Context Structure context for a property handle.

SetResult - Set Result

The function stores the result of a context function in the result area, which can be retrieved in the application using the PropertyHandle::GetResult() function.

```
void CTX_Structure :: SetResult (char *result_string )
```

result_string	Result string The result string can be a list of strings where strings are usually separated by x01 characters. If there is only one string returned the string is 0-terminated. Multiple strings are terminated with 0 after the last string in the list, which should be terminated with x01 as well.

SetTransactionError - Set Transaction Error

The function marks a transaction as errounus. This leads to a rollback of the transaction when the transaction is finished. The function can be called in post handlers (as inserted or deleted) to undo the performed operation.

```
void CTX_Structure :: SetTransactionError ( )
```

ShowInstance - Show instannce

The show instance function resets the hidden state for an instance. Usually, the hidden state is reset automatically, when the selection in a property handle is changed.

The function returns the 'hidden' state as it was set before calling the function.

```
logical CTX_Structure :: ShowInstance ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

~CTX_Structure - Destructor

The function destroys the structure context. The function must be overloaded in an application specific implementation of the structure context.

```
CTX_Structure :: ~CTX_Structure ( )
```

DBErrorHandle - Database Error Handle

The database error handle provides extended documentation for errors detected in the system. In contrast to the basic ErrorHandle the DBErrorHandle locates signaled errors in the system or application database and provides detailled information for the error detected.

A application specific error handle can be defined and set for enabling application specific error handling (-> ErrorHandle).

DBErrorHandle - Constructoe

The cnstructor creates an error handle. The object_handle passed to the function should contain the error descriptions in an extent as defined in the error(s) that are handled by the error handler. For system errors this is a database object handle for the system database ode.sys. For application errors the resource database or dictionary should contain the error definitions. To activate an error handler for the errors of a certain error class you can use the Error::SetErrorHandle() function.

```
DBErrorHandle :: DBErrorHandle (DBObjectHandle &object_handle )
```

object_handle	Database Object handle This is a pointer to an opened Database Object handle.

DisplayMessage - Display message

The function displays a message to the console or in a message box (when GUI-messages are activated). Displaying messages can be supressed by setting the SUPRESS_ERRORS system variable to "YES". Besides writing the message to the protocol file it will be displayed on console (for console applications). This can be supressed by setting the system variable NO_CONSOLE_MESSAGES to "YES".

```
logical DBErrorHandle :: DisplayMessage (Error *error_obj )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

error_obj

GetError - Get error text from resource database

The function provides the error text stored in the resource data base passed as error_source or set as error_source in the error object. This is usually the application resource database that contains the error defintions in an extent with the name of the error class. The error text will be provided according to the language set in the error handle.

When the error is not found a message "Undefined error" containing the error variables set is created. The function returns the error type as set in the error defintion.

```
char DBErrorHandle :: GetError (Error *error_obj, void
                         *error_source )
```

Return value

error_obj

error_source

GetErrorHelpID - Get context help id for the error

The function returns a context help id for the error that can be used to call the online help for errors, that can be created using the ODABA design tools or by any other application. By default the error context id is the resource id of the error definition.

```
int32 DBErrorHandle :: GetErrorHelpID (Error *error_obj )
```

Return value

error_obj

GetObjectHandle - Get resource object handle

The function returns the database object handle for the error handle that has been set for error look up.

```
DBObjectHandle &DBErrorHandle :: GetObjectHandle ( )
```

Return value — This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.

SetObjectHandle - Set resource object handle

The function sets the database object handle for the error handle that is used for error look up.

```
void DBErrorHandle :: SetObjectHandle (ACObject *obhandle )
```

obhandle	Database Object Handle This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.

~DBErrorHandle - Destructor

The function destroys the error handle.

```
                        DBErrorHandle :: ~DBErrorHandle ( )
```

DBFieldDef - Definition for the internal presentation of property data

The internal property definition contains all information available and necessary accessing data of the property. Among basic information such as type and size it contains special ODABA2 access information such as index and base collection definitions.

Alls these information are used for reading and writing data just as to execute operations on properties (see also **{.r DBField**}).

DBFieldDef - Constructor

i0

```
          DBFieldDef :: DBFieldDef (char
*fldnames, char *fldtypes, SDB_RLEV fldreflev,
uint16 fldsize, uint16 fldprec, uint16 flddim,
smcb *smcbptr, char *gentype, logical secrefr,
char *extnames, char *irefname, DBIndex
*indexptr )
```

fldnames	
fldtypes	
fldreflev	
fldsize	
fldprec	
flddim	
smcbptr	Pointer to generel structure definition The smcb is a more general way to define structure (DBStructDef). It contains information for the structure and its properties. In contrast to the DBStructDef the smcb describes structure members regardless on the rule they may play in the structure.
gentype	Generic type of property Pointer to a null-terminated string containing the generic type.
secrefr	Property is secundary referenced

extnames	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
irefname	Name of inverse property
indexptr	Index definition Pointer to an internal index definition.

i04

```
DBFieldDef :: DBFieldDef (fmcb *fmcbptr )
```

fmcbptr

i1

```
DBFieldDef :: DBFieldDef ( )
```

i2

```
DBFieldDef :: DBFieldDef (Dictionary *dictptr, SDB_Reference *dbrptr, smcb *smcbptr, DBIndex *indexptr, logical domopt, logical logrefr, logical secrefr, logical depopt, char *extnames, char *irefname )
```

dictptr	Dictionary handle An opened dictionary handle is passed.
dbrptr	ODABA2 reference definition Pointer to a reference definition instance, stored in an ODABA2 data base.
smcbptr	Pointer to generel structure definition The smcb is a more general way to define structure (DBStructDef). It contains information for the structure and its properties. In contrast to the DBStructDef the smcb describes structure members regardless on the rule they may play in the structure.
indexptr	Index definition Pointer to an internal index definition.
domopt	Property is dominant
logrefr	Property is a logical reference

secrefr	Property is secundary referenced
depopt	Property data depends on property
extnames	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
irefname	Name of inverse property

i3

```
                         DBFieldDef :: DBFieldDef (SDB_Property
                  *dbyptr, smcb *smcbptr )
```

dbyptr	
smcbptr	Pointer to generel structure definition The smcb is a more general way to define structure (DBStructDef). It contains information for the structure and its properties. In contrast to the DBStructDef the smcb describes structure members regardless on the rule they may play in the structure.

GetDBStruct - Get Database structure definition

The function returns the database structure definition for the field. When the field is not associated with a database definition or when the associated structure definition is an smcb, only, and not a database structure definition the function returns 0.

i0

```
DBStructDef *DBFieldDef :: GetDBStruct ( )
```

Return value

i1

```
DBStructDef *DBFieldDef :: GetDBStruct (Dictionary *dictptr )
```

Return value	
dictptr	Dictionary handle An opened dictionary handle is passed.

i2

```
DBStructDef *DBFieldDef :: GetDBStruct (Dictionary *dictptr,
                     uint8 schemaversion )
```

Return value	
dictptr	Dictionary handle An opened dictionary handle is passed.
schemaversion	Scheme version Number of version for the scheme of data structure definitions. If the number is not known, it can be retrieved from the {.r Dictionary} via the function {.r ACObject.GetVersion}().

GetExtendName -

```
char *DBFieldDef :: GetExtendName ( )
```

Return value	The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

GetIndexDef -

i0

```
DBIndex *DBFieldDef :: GetIndexDef (int16 indx_pos )
```

Return value	
indx_pos	

i1

```
DBIndex *DBFieldDef :: GetIndexDef (char *keyname )
```

Return value	
keyname	Name of sort key The order key name must be a key name defined for the given structure. The sort key is passed as 0-terminated string with maximum 40 characters.

IsBaseCollection -

```
logical DBFieldDef :: IsBaseCollection (Dictionary *dictptr,
                          char *strnames )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
dictptr	Dictionary handle An opened dictionary handle is passed.
strnames	Structure name The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.

get_clst_table - Instances of property collection are stored in a cluster

This characteristic is of interest for reference collections, only.

The function returns YES, if all instances of the collection are stored in a cluster.

```
logical DBFieldDef :: get_clst_table ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

get_create - Property is allowed to create new instances

This characteristic is of interest for references and relationships, only.

The function returns YES, if new instances can be created via this property.

```
logical DBFieldDef :: get_create ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

get_depend - Instance(s) depends on the relationship

This characteristic is of interest for relationships, only.

The function returns YES, if instances are deleted immedialety when removing from the relationship.

```
logical DBFieldDef :: get_depend ( )
```

Return value	The value YES means that all instances referenced by the relationship depends on the relationship and will be deleted, when they are removed from the relationship (see also {.r SDB_Relationship.depend}).

get_extend -

```
const char *DBFieldDef :: get_extend ( )
```

Return value	The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

get_gen_type -

```
const char *DBFieldDef :: get_gen_type ( )
```

Return value	Pointer to a null-terminated string containing the generic type.

get_initval -

```
const char *DBFieldDef :: get_initval ( )
```

Return value

get_inverse -

```
DBFieldDef *DBFieldDef :: get_inverse ( )
```

Return value

get_inverse_name -

```
const char *DBFieldDef :: get_inverse_name ( )
```

Return value

get_mb_number -

```
int16 DBFieldDef :: get_mb_number ( )
```

Return value	Mainbase numbers from 0 to 252 (for small databases) and 0 to 32767 (for large databases) are valid.

get_multikey -

```
logical DBFieldDef :: get_multikey ( )
```

Return value

get_owning -

```
logical DBFieldDef :: get_owning ( )
```

Return value

get_privilege -

```
PIADEF DBFieldDef :: get_privilege ( )
```

get_static -

```
logical DBFieldDef :: get_static ( )
```

get_transient -

```
logical DBFieldDef :: get_transient ( )
```

get_update -

```
logical DBFieldDef :: get_update ( )
```

Return value

get_version -

```
int16 DBFieldDef :: get_version ( )
```

Return value

get_virtual -

```
logical DBFieldDef :: get_virtual ( )
```

Return value

get_weak_typed -

```
logical DBFieldDef :: get_weak_typed ( )
```

Return value

operator=

```
logical DBFieldDef :: operator= (DBFieldDef &dbfield_ref )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbfield_ref	

set_initval

```
void DBFieldDef :: set_initval (char *init_string )
```

init_string	Initial value The initial value for the property is passed as 0-terminated string.

~DBFieldDef - Destructor

```
DBFieldDef :: ~DBFieldDef ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DBObjectHandle - Database Object handle

Database object handles are necessary for accessing data in an database object. A database object can be considered as a database within a database. Each database has a root database object on top. Below each database object any number of subordinated database objects can be created.

Database objects in a database are logically separated but not physically. Thus, it becomes possible to establish links between structure instances in different database objects. Each database object has, however, its own extents containing the global instances of the database object.

The database object handle for the root database object is part of the database handle (-> DatabaseHandle) and need not to be opened explicitly.

A database object handle is required for opening extent property handles for accessing structure instances stored in extents.

The database object handle administrates transactions. Transactions can be started and stopped for each object handle. The database object handle is not thread save, i.e. a database object handle must not be used simultaneously in different threads.

The database object supports version slices, i.e. each database object may have its own current version.

BeginTransaction - Start transaction

When starting a transaction all modification for the database are stored in a transaction buffer. A transaction can start as an internal or external transaction.

Internal transactions are used for small transactions upto 10000 updated database entries. Usually a small transaction takes just a few seconds. By defining a maximum buffer count for the transaction you can define a dynamical transaction buffer for speeding up processes as copying data. In this case the transaction buffer will be cleared automatically when the buffer limit is reached.

External transactions are stored in a transaction database which is created in a path defined in the TABASE system variable (or ini-file variable). External transactions are a little bit slower than internal ones but they are not limited in capacity.

Transactions can be nested. When starting a transaction while another transaction is running the new transaction creates a transaction within a transaction. The nesting level (>0) is returned as transaction level. When the function returns 0 the transaction could not be started.

Entries, wich are stored in a transaction are locked for other users until the top-transaction has been terminated.

Updates can be moved to the upper transaction or stored in the database using CommitTransaction(). Only commiting the top transaction will store the updates to the database. Updates made within a transaction become visible in an upper transaction when the transaction is closed. They become visible for other users when the top transaction is closed (CommitTransaction).

RoleBack() can be used to undo all updates made within a transaction.

```
int16 DBObjectHandle :: BeginTransaction (logical ext_TA, int16
                          w_maxnum )
```

Return value	The transaction level is usually 1.For nested transaction it corresponds to the nesting level.
ext_TA	External transaction

YES must be passed to start the transaction as external transaction, i.e. all modification are stored to an external transaction base. Otherwise (NO) the transaction is started as internal transaction, i.e. the modifications are stored in memory.

w_maxnum	Maximum number of entries in transacktion (buffer size)
	The maximum number should be set to UNDEF (0) for indicating to save only the complete transaction. For defining a transaction buffer to optimize read/write options use the maximum number of transaction buffer entries (e.g. 300).

ChangeTimeStamp - Change time stamp for current version

Each version for a database object has a final time limit. As long as the time limit for the version lies in the future you can change the version end by setting a new time stamp. The new time stamp must always lie in the future.

```
logical DBObjectHandle :: ChangeTimeStamp (uint16 version_nr,
                              dttm timestamp )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
version_nr	Internal version number
	Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version.
	Default: CUR_VERSION
timestamp	Timestamp
	A time stamp defines a time point by passing date and time.

Close - Close Object Handle

Closing the database object handle will reduce the use count for the access block. The database object access block is removed, when the use count becomes 0, i.e. when the last database object handle referring to this resource is closed or destroyed.

```
logical DBObjectHandle :: Close ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

CommitTransaction - Commit transaction

The function stores all changes made in the transaction to the higher transaction level. When terminating a transaction for a given transaction level transactions on lower levels are commited as well. In contrast to other functions CommitTransaction will not reset the error, i.e. after committing the transaction any error or warning set during the transaction is still set.

The function returns an error (YES), when the transaction could not be stored because of an error. This may happen when a top transaction tries to write to the database or when the transaction has set an error within the transaction that does not allow storing the transaction. In this case the error signaled while committing the transaction will overwrite any previously set error.

When ppassing AUTO (-1) as transaction level, the current (last recently opened) transaction will be closed.

```
logical DBObjectHandle :: CommitTransaction (int16 talevel )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
talevel	Transaction level The transaction level is usually 1.For nested transaction it corresponds to the nesting level.

DBObjectHandle - Create an Database Object handle

The function creates a new database object handle.

ci - Create database object handle

The constructor creates a database object handle from a database handle. The constructor creates a new access block for the database object handle, that refers to the root object of the database handle.

```
          DBObjectHandle :: DBObjectHandle (DBHan-
dle *dbhandle, PIACC accopt, uint16 ver-
sion_nr, ResourceTypes local_ressources )
```

dbhandle	Pointer to database handle This is pointer to an opened database handle. The database handle can be an opened database handle DataBaseHandle as well as an opened dictionary handle (DictionaryHandle).
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).
version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION
local_ressources	Resource type

Depending on the resource type the database or dictionary is opened on the client or server side.

RES_automatic

When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side.

RES_local

The dictionary or database will be opened on the client machine in any case.

RES_server

The dictionary or database will be opened on the server machine side in any case.

ci1 - Create sub-ordinated object handle by identity

The constructor creates a sub-ordinated object handle by locating the database object using the database object identification number. A new access block is created and associated with the database object handle.

```
DBObjectHandle :: DBObjectHandle (DBObjectHandle &dbobject, int32 objid, PIACC accopt, uint16 version_nr, ResourceTypes local_ressources )
```

dbobject	Database Object handle
objid	Local object identity (LOID) The local object identity is a 31-bit number that identifies an object instance uniquely in a database. LOIDs are available only for independently stored instances but not for instances of imbedded structures.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).
version_nr	Internal version number

Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version.

Default: CUR_VERSION

local_ressources	Resource type

Depending on the resource type the database or dictionary is opened on the client or server side.

RES_automatic

When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side.

RES_local

The dictionary or database will be opened on the client machine in any case.

RES_server

The dictionary or database will be opened on the server machine side in any case.

ci2 - Create sub-ordinated object handle by name

The constructor creates a sub-ordinated object handle by locating the database object using the database object name. A new access block is created and associated with the database object handle.

```
DBObjectHandle :: DBObjectHandle (DBObjectHandle &dbobject, char *objname, PIACC accopt, uint16 version_nr, ResourceTypes local_ressources )
```

dbobject	Database Object handle
objname	Database object name
	Database object name is passed as 0-terminated string with maximum 40 charcters.
accopt	Access option

The access option defines the way instances in a property handle are to be accessed (read, update, write).

version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION
local_ressources	Resource type Depending on the resource type the database or dictionary is opened on the client or server side. **RES_automatic** When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side. **RES_local** The dictionary or database will be opened on the client machine in any case. **RES_server** The dictionary or database will be opened on the server machine side in any case.

ci3 - Create database object for data source

This constructor creates a database object handle according to the specification in a data source. The passed access mode allows overwriting the access mode defined in the data source. The data source must be described either in the ini-file passed to the application or in the data catalogue defined in the ini-file.

```
DBObjectHandle :: DBObjectHandle (ODABAClient &odaba_client, char *data_source_name, PIACC access_mode, ResourceTypes local_ressources )
```

odaba_client	ODABA Client Handle

The ODABA client handle can be passes as connectet or ea empty handle.

data_source_name	Data source name The data source name is passed as 0-terminated string with a maximum length of 40 characters.
access_mode	Access mode The access option defines the way instances in a property handle are to be accessed (read, update, write). Default: PI_Read
local_ressources	Resource type Depending on the resource type the database or dictionary is opened on the client or server side. **RES_automatic** When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side. **RES_local** The dictionary or database will be opened on the client machine in any case. **RES_server** The dictionary or database will be opened on the server machine side in any case.

i05 - Copy constructor

The function creates a database object handle that uses the same access block as the handle passed to the constructor.

```
                DBObjectHandle :: DBObjectHandle (const
DBObjectHandle &obhandle_refc )
```

obhandle_refc	Const reference to database object handle The reference refers to an opened or not opened database object handle.

i3 - Internal constructor

This constructor is used internally, only.

```
                         DBObjectHandle :: DBObjectHandle (ACOb-
                  ject *acobject )
```

acobject

i4 - Dummy constructor

The constructor creates a database object handle without access block. This handle cannot be used until it opened explicitly using the Open() function.

```
                         DBObjectHandle :: DBObjectHandle (
                  )
```

DisableEventHandling - Disable event handling

The function will disable external event handlingfor the database (object), i.e. events are not sent to external event handlers set for property handles or to the client.

```
void DBObjectHandle :: DisableEventHandling (
                  )
```

EnableEventHandling -

The function will enable external event handling for the database (object) after it has been disabled, i.e. events are sent again to external event handlers set for property handles or to the client.

```
void DBObjectHandle :: EnableEventHandling ( )
```

EventHandling - Is event handling enabled?

The function returns whether external events are enabled or not (see EnableEventHandling() and DisableEventHandling())

```
logical DBObjectHandle :: EventHandling ( )
```

Return value The function returns YES when the question was answered positivly. Otherwise it returns NO.

ExecuteDBObjectAction - Execute object context function

The function calls an action that is defined in the database object context. The function is executed on the server side first. If it was executed successfully, the function is executed on the client side, too.

The action may use the SetActionResult() function to pass the result of the action to the client application. If execution of the function on the client side returns NO the result passed from the server overwrites any result set by the client function. The result can be retrieved from the client application using the function GetActionResult().

```
logical DBObjectHandle :: ExecuteDBObjectAction (char
                *action_name, char *parm_string )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
action_name	Name of the action to be performed The name of the action is passed as 0-terminated string with a maximum length of 40 significant characters.
parm_string	Parameter string The parameter string is passed as 0-terminated string and contains the parameters according to the conventions of the action called.

ExtentExist - Does Extent exist in database object

An extent defined logically in the database schema need not exist in a database or database object. Extents are created in the database object automatically when accessing it the first time with write access. The function returns whether an extent has been already created in the given object (YES) or not (NO).

```
logical DBObjectHandle :: ExtentExist (char *extnames )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
extnames	Extent name

The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

GetAccess - Get access mode for object handle

The function returns the access mode set for the object handle when opening it.

```
PIACC DBObjectHandle :: GetAccess ( )
```

Return value — The access option defines the way instances in a property handle are to be accessed (read, update, write).

GetActionResult - Get result from last action executed

The function returns the resultstring from the last action executed. The result string is available until the next action call, only. When the action does not return a result the function returns NULL.

```
char *DBObjectHandle :: GetActionResult ( )
```

Return value — The result string can be a list of strings where strings are usually separated by x01 characters. If there is only one string returned the string is 0-terminated. Multiple strings are terminated with 0 after the last string in the list, which should be terminated with x01 as well.

GetDBHandle - Provide database handle

The function returns the database handle the for the database the referenced database object belongs to.

```
DatabaseHandle &DBObjectHandle :: GetDBHandle ( )
```

Return value — This is pointer to an opened database handle. The database handle can be an opened database handle DataBaseHandle as well as an opened dictionary handle (DictionaryHandle).

GetDictionary - Get dictionary handle

The function returns the dictionary handle for the database.

```
DictionaryHandle &DBObjectHandle :: GetDictionary ( ) const
```

Return value — An opened dictionary handle is passed.

GetExtent - Provide extent form Database Object

The function returns the name of the n-th extent in the list of extents that are allocated for the database object. The collection of allocated extents does not necessarily include all defined extents. Extent names are provided in alphabetic order. The first extent has the index 0.

After providing the last extent name the function returns NULL for the next extent name.

```
char *DBObjectHandle :: GetExtent (int32 indx0 )
```

Return value	The name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
indx0	Position in collection The position in the collection addresses the first instance in the collection with 0. AUTO (-1) refers to an undefined position.

GetHighObject - Get parent object

The function returns the parent database object handle.

```
DBObjectHandle &DBObjectHandle :: GetHighObject ( )
```

Return value	This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.

GetObject - Get Database Object

The function returns the name of the n-th database object in the list of sub-ordinated objects. Database object names are provided in alphabetic order. The first object has the index 0.

After providing the last database object name the function returns NULL.

```
char *DBObjectHandle :: GetObject (int32 indx0 )
```

Return value	Database object name is passed as buffer with 40 charcters.
indx0	Position in collection

The position in the collection addresses the first instance in the collection with 0. AUTO (-1) refers to an undefined position.

GetStructDef - Get structure definition

The function returns the structure definition for the structure name passed. The structure definition is provided from the dictionary associated with the database object handle.

```
DBStructDef *DBObjectHandle :: GetStructDef (char *strnames )
```

Return value	The structure definition (DBStructDef) contains the metadata for the instance, i.e. information for the structure and its properties.
strnames	Structure name The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.

GetSystemVersion - Get system version

The fiunction provides the schema version of the ODABA system, which is the dictionary for a dictionary.

```
uint16 DBObjectHandle :: GetSystemVersion ( )
```

Return value	Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION

GetTimeStamp - Get date/time for version

The function returns the termination time for the version number passed to the function.

```
dttm DBObjectHandle :: GetTimeStamp (uint16 version_nr )
```

Return value	A time stamp defines a time point by passing date and time.
version_nr	Internal version number

Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version.

Default: CUR_VERSION

GetVersion - Get version number for the time point

The function returns the veriosn number that includes the passed time point.

```
uint16 DBObjectHandle :: GetVersion (dttm timestamp )
```

Return value	Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION
timestamp	Timestamp A time stamp defines a time point by passing date and time.

IsClient - Is database object client object?

The function returns, whether the database object has been created on the client side (YES) or not (NO). Database objects in local applications are both, client and server objects.

```
logical DBObjectHandle :: IsClient ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsOpened - Is database object opened?

The function returns whether the database object has been opened (YES) or not (NO), i.e. whether an access block is asociated with the handle.

```
logical DBObjectHandle :: IsOpened ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsValid - Is database object valid?

The function returns whether the database object has been opened and whether the associated access block is valid (YES) or not (NO).

```
logical DBObjectHandle :: IsValid ( ) const
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

NewVersion - Create new version

The function creates a new version slice for the database object. The timestamp passed indicates, when the current version is to be closed and when the new version will start. You cannot define a timepoint in the passed for ctrating a new version, i.e. the time point must be 'now' (empty) or a value that is in the future.

New versions can be created for databases enabled for workspaces only, when all workspaces are empty (consolidated or discarded).

```
logical DBObjectHandle :: NewVersion (dttm timestamp, uint16
                          version_nr )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
timestamp	Timestamp A time stamp defines a time point by passing date and time.
version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION

Open - Open database object handle

The function opens a database object handle. When an access block is opened for the object handle it will be closed before.

ci - Open database object handle

The function opens a database object handle for the database handle. The function creates a new access block for the database object handle, that refers to the root object of the database handle.

```
logical DBObjectHandle :: Open (DBHandle *dbhandle, PIACC ac-
                  copt, uint16 version_nr, ResourceTypes lo-
                  cal_ressources )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbhandle	Pointer to database handle This is pointer to an opened database handle. The database handle can be an opened database handle DataBaseHandle as well as an opened dictionary handle (DictionaryHandle).
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).
version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION
local_ressources	Resource type

Depending on the resource type the database or dictionary is opened on the client or server side.

RES_automatic

When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side.

RES_local

The dictionary or database will be opened on the client machine in any case.

RES_server

The dictionary or database will be opened on the server machine side in any case.

ci1 - Open sub-ordinated object handle by identity

The function opens a sub-ordinated object handle by locating the database object using the database object identification number. A new access block is created and associated with the database object handle.

```
logical DBObjectHandle :: Open (DBObjectHandle &dbobject, int32
                  objid, PIACC accopt, uint16 version_nr, Re-
                  sourceTypes local_ressources )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbobject	Database Object handle
objid	Local object identity (LOID) The local object identity is a 31-bit number that identifies an object instance uniquely in a database. LOIDs are available only for independently stored instances but not for instances of imbedded structures.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).

version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION
local_ressources	Resource type Depending on the resource type the database or dictionary is opened on the client or server side. **RES_automatic** When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side. **RES_local** The dictionary or database will be opened on the client machine in any case. **RES_server** The dictionary or database will be opened on the server machine side in any case.

ci2 - Open sub-ordinated object handle by identity

The function opens a sub-ordinated object handle by locating the database object using the database object name. A new access block is created and associated with the database object handle.

```
logical DBObjectHandle :: Open (DBObjectHandle &dbobject, char
                *objname, PIACC accopt, uint16 version_nr, Re-
                sourceTypes local_ressources )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbobject	Database Object handle
objname	Database object name

	Database object name is passed as 0-terminated string with maximum 40 charcters.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).
version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION
local_ressources	Resource type Depending on the resource type the database or dictionary is opened on the client or server side. **RES_automatic** When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side. **RES_local** The dictionary or database will be opened on the client machine in any case. **RES_server** The dictionary or database will be opened on the server machine side in any case.

ci3 - Open Database Object Handle for Data Source

This function opens a database object handle according to the specification in a data source. The passed access mode allows overwriting the access mode defined in the data source. The data source must be described either in the ini-file passed to the application or in the data catalogue defined in the ini-file.

```
logical DBObjectHandle :: Open (ODABAClient &odaba_client, char
                 *data_source_name, PIACC access_mode, Re-
                 sourceTypes local_ressources )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
odaba_client	ODABA Client Handle The ODABA client handle can be passes as connectet or ea empty handle.
data_source_name	Data source name The data source name is passed as 0-terminated string with a maximum length of 40 characters.
access_mode	Access mode The access option defines the way instances in a property handle are to be accessed (read, update, write). Default: PI_Read
local_ressources	Resource type Depending on the resource type the database or dictionary is opened on the client or server side. **RES_automatic** When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side. **RES_local** The dictionary or database will be opened on the client machine in any case. **RES_server** The dictionary or database will be opened on the server machine side in any case.

RollBack - Roll back transaction

The function discards all changes made in the transaction. When terminating a transaction for a given transaction level transactions all lower levels are discarded as well. In contrast to other functions RollBack will not reset the error, i.e. after rolling back the transaction an error set during the transaction is still set.

The function returns an error (YES), when the transaction could not be reset because of an error. This may happen when some of the included access blocks could not be reset properly. In this case the error signaled during roll back of transaction will overwrite an error set during the transaction.

```
logical DBObjectHandle :: RollBack (int16 talevel )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
talevel	Transaction level The transaction level is usually 1.For nested transaction it corresponds to the nesting level.

SetActionResult - Set result string

The function allows setting a result string for the database object (or the database) handle. The result string can be retrieved with the GetActionResult function. Thus you can pass the result of any action also to a client application while the action is running on the server. The result is passed as string, i.e. the result must not contain any 0-characters except the terminating 0.

```
void DBObjectHandle :: SetActionResult (char *result_string )
```

result_string	Result string The result string can be a list of strings where strings are usually separated by x01 characters. If there is only one string returned the string is 0-terminated. Multiple strings are terminated with 0 after the last string in the list, which should be terminated with x01 as well.

SetOverload - Set object overload

The function activates the object overload feature. When this option is activated, extents in the current database object will overload extents with the same name in parent objects. An extent contains the objects from all instances allong the database object hierarchy.

```
char DBObjectHandle :: SetOverload (logical overload_opt )
```

Return value	When this option is set to yes extents in a object hierarchy can be overloaded.
overload_opt	Overload option When this option is set to yes extents in a object hierarchy can be overloaded.

SetServerVariable - Set system variable on server

Systemvariables can be set for the server. This is necessary for controlling functions running on the server side.

Server variables are valid on the server only for the connected client.

```
logical DBObjectHandle :: SetServerVariable (char *var_name,
                                  char *var_string )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
var_name	System variable name Name of the system variable on the server or client side. System variable names must not exceed 40 characters and are provided as 0-terminated strings.
var_string	Value for the system variable The value for a system variable must not exceed 255 characters and is provided as 0-terminated string.

SetVersion - Set current version

The function sets the current version slice that should be active when accessing instances in the object. Since the function may influence data selected in property handles all property handles should be saved and cancelled before calling the function. Data might get incompatible when accessing another version and must be refreshed if not cancelled.

i0 - Set verion according to number

The current database version is set according to the passed version number. The version number must be less or equal to the last version number created for the database object.

```
logical DBObjectHandle :: SetVersion (uint16 version_nr )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION

i1 - Set verion according to date

The current database version is set according to the passed date, i.e. to the version slice (version number) that includes the passed date. The date should be the current date or a darte in the passed.

```
logical DBObjectHandle :: SetVersion (dbdt date )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
date	Date

i2 - Set verion according to timestamp

The current database version is set according to the passed timepoint, i.e. to the version slice (version number) that includes the passed timepoint. The date should be the current date or a darte in the passed.

```
logical DBObjectHandle :: SetVersion (dttm timestamp )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
timestamp	Timestamp A time stamp defines a time point by passing date and time.

VersionCount - Get number of versions

The function returns the number of versions defined for the database object. The number is identical with the last version number defined for the database object. When no version has created for the database object the function returns 0.

```
int32 DBObjectHandle :: VersionCount ( )
```

Return value

VersionIntervall - Get version interval

The function returns the version interval, i.e. the begin and end of the version slice with the passed version number.

```
INTERVAL(dttm) DBObjectHandle :: VersionIntervall (uint16 ver-
                 sion_nr )
```

Return value	The time interval contains two timepoints (DATETIME) for begin and end of the time interval.
version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION

operator bool - DBObjectHandle opened?

The function returns YES (true) when the database object is opened and NO (false) when the database object is not opened or when an error had occured while constructing the dictionary handle.

```
NOTYPE DBObjectHandle :: operator bool ( ) const
```

Return value

operator= - Assigning a database object handle

The function will close the odatabase object handle, when it is opened. The access block from the passed database object handle is associated with the current handle increasing the use count by 1.

```
DBObjectHandle &DBObjectHandle :: operator= (const DBObjectHan-
                  dle &obhandle_refc )
```

Return value	This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.
obhandle_refc	Const reference to database object handle The reference refers to an opened or not opened database object handle.

operator== - Are handles using the same access blocks?

The operator returns whether the handles refer to the same database object, i.e. to the same access block.

i0

```
logical DBObjectHandle :: operator== (const DBObjectHandle
                  &obhandle_refc )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
obhandle_refc	Const reference to database object handle The reference refers to an opened or not opened database object handle.

i01

```
logical DBObjectHandle :: operator== (const DatabaseHandle
                          &dbhandle_refc )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

i02

```
logical DBObjectHandle :: operator== (const DictionaryHandle
                          &dictionary_refc )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

~DBObjectHandle - Destructor

The destructor closes the database object handle. Closing the database object handle will reduce the use count. The internal resources, the database object access block is removed, when the use count becomes 0, i.e. when the last database object handle referring to this resource is closed or destroyed.

```
                                    DBObjectHandle :: ~DBObjectHandle ( )
```

DBStructDef - Definition for the internal presentation of data structures and enumerations

Definitions for data structures are usually read from an ODABA2 dictionary. However they can be provided and filled directly in main storage. Still in this case the definition should be provided via Dictionary functions to make them available for the ODABA2 kernel.

From an ODABA2 dictionary structures are provided only, if they are marked as checked and as ready for a non test environment.

DBStructDef - Constructor

i0

```
DBStructDef :: DBStructDef (char
*strnames, int16 strsid, int32 intlen, int32
extlen, TYP_TYPES metatype, SDB_ST strtype,
int16 basecount, int16 attrcount, int16 re-
frcount, int16 rshpcount, char *idkeynames,
logical w_vf_opt, int16 w_schema_version, log-
ical w_versioning, logical glob_identity )
```

strnames	Structure name The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.
strsid	Internal structure number Internal number identifying a data structure within a ODABA2 dictionary.
intlen	Internal length
extlen	External length
metatype	Meta type for a type definition Via a structure definition can described a data structure, a basic data type or an enumeration.
strtype	Structure meta type
basecount	Number of base structures
attrcount	Number of attributes

refrcount	Number of references
rshpcount	Number of relationships
idkeynames	Name of the identifying key Pointer to a null-terminated string containing the ident key name.
w_vf_opt	Consider virtual function pointer
w_schema_version	Scheme version Number of version for the scheme the data structure definitions stands for. If the number is not known, it can be retrieved from the {.r Dictionary} via the function {.r ACObject.GetVersion}().
w_versioning	Consider online versioning for data

i01

```
DBStructDef :: DBStructDef (
)
```

GetAttrPath - Provide path for the indexed attribute

The function retrieves the path for attributes with basic types, only. Structured Attributes referenced directly (no pointers) such as base structures are resolved.

Generic attributes are considered as references in this case (see also **{.r DBStructDef.GetRefPath()**}).

```
int32 DBStructDef :: GetAttrPath (int32 indx0, logical
                     full_path, char *fld_path, int32 maxlen, logi-
                     cal with_generics, logical search_in_sharebase
                     )
```

Return value	Position the entry is located at. If the entry could not be locates its contains the number of entries for the data structure.
indx0	Position in collection The position in the collection addresses the first instance in the collection with 0. AUTO (-1) refers to an undefined position.
full_path	Full path

	The full path option is used to request the property path including base structure names.
	Default: YES
fld_path	Property path
	Pointer to the buffer that is to receive the property path string.
maxlen	Size of output buffer
	Specifies the length of the buffer, the information should be stored into. The information is truncated if it is longer than the buffer.

GetEntry - Provide DB-FieldDefinition entry

The function retrieves the definition for a property by name or position.

Using this function, only property definitions explicitely defined for this structure can be retrieved.

Use **{.r smcb.SearchField()**} to retrieve a property from a base structure or from a structured attribute vie path.

i0

```
DBFieldDef *DBStructDef :: GetEntry (char *fldnames )
```

Return value

fldnames

i1

```
DBFieldDef *DBStructDef :: GetEntry (int16 sindex )
```

Return value

GetRefPath - Provide path for the indexed reference

The function retrieves the path for references and relationships in base structures, for generic attributes attributes and for references and relationships of the structure itself.

```
int32 DBStructDef :: GetRefPath (int32 indx0, logical full_path,
                 char *fld_path, int32 maxlen, logical
                 with_generics, logical search_in_sharebase )
```

Return value	Position the entry is located at. If the entry could not be locates its contains the number of entries for the data structure.
indx0	Position in collection The position in the collection addresses the first instance in the collection with 0. AUTO (-1) refers to an undefined position.
full_path	Full path The full path option is used to request the property path including base structure names. Default: YES
fld_path	Property path Pointer to the buffer that is to receive the property path string.
maxlen	Size of output buffer Specifies the length of the buffer, the information should be stored into. The information is truncated if it is longer than the buffer.

GetSortKeySMCB - Provide key definition

```
smcb *DBStructDef :: GetSortKeySMCB (char *fldnames )
```

Return value	The smcb is a more general way to define structure (DBStructDef). It contains information for the structure and its properties. In contrast to the DBStructDef the smcb describes structure members regardless on the rule they may play in the structure.
fldnames	

GetStrDefVersion - Provide structure definition for a previous scheme version

The function retrieves the data structure definition valid for the given scheme version. Usually this structure definition is read from the dictionary.

If the scheme version is invalid the function returns NULL.

```
DBStructDef *DBStructDef :: GetStrDefVersion (Dictionary
                    *dictptr, uint8 schemaversion )
```

dictptr	Dictionary handle An opened dictionary handle is passed.
schemaversion	Scheme version Number of version for the scheme of data structure definitions. If the number is not known, it can be retrieved from the {.r Dictionary} via the function {.r ACObject.GetVersion}().

IsBasedOn - Is the data structure a specialization of another one ?

The function returns YES, if the data structure has a base structure of given type. The base structure is searched recursive.

```
logical DBStructDef :: IsBasedOn (char *strnames )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
strnames	Structure name The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.

get_attr_info - Provide information about the attribibutes of the data structure

```
MemberInfo &DBStructDef :: get_attr_info ( )
```

Return value

get_base_info - Provide information about the base structures of the data structure

```
MemberInfo &DBStructDef :: get_base_info ( )
```

Return value

get_refr_info - Provide information about the references of the data structure

```
MemberInfo &DBStructDef :: get_refr_info ( )
```

Return value

get_rshp_info - Provide information about the relationships of the data structure

```
MemberInfo &DBStructDef :: get_rshp_info ( )
```

Return value

get_sb_number - Provide SubBase number

Persistant instances of a data structure can be stored in a defined SubBase within the ODABA2 data base (see also **{.r SDB_ODABA_Str.sb_number**}).

The function retrieves the SubBase Number defined for the data structure.

```
int16 DBStructDef :: get_sb_number ( )
```

Return value	Sub-bases for a main base are numbered contineously. The highest sub-base number is 255.

get_schema_ver - Provide the scheme version

The function retrieves the scheme version of the data structure definition.

Usually this is the number of the project version, within the data structure was modified last time.

```
int16 DBStructDef :: get_schema_ver ( )
```

Return value	Number of version for the scheme of data structure definitions. If the number is not known, it can be retrieved from the {.r Dictionary} via the function {.r ACObject.GetVersion}().

~DBStructDef - Destructor

```
DBStructDef :: ~DBStructDef ( )
```

DB_Event - Database Events

Database events are generated when the process state of an instance or collection changes (e.g. open or read) or when general changes on an instance or collection are signaled. Database events are signaled, when performing special system actions as a requested action did not perform well (e.g. not deleted).

There are three different groups of events. Pre-events or process events (DBP_...) are generated before running a requested action. When handling those events the event handler can return an error (YES) to abort the action. This allows checking the action to be performed. Post-events (DBO_...) are events that are generated after performing the action. The return code from a post event is not checked by the system. Post-events allow application specific actions after the action has been performed. Error events (DBO_Not...) are signaled when the action aborted because of the returncode of the pre-handler or because of an error.

Events are generated for instances and properties (collections, attributes). Database events can be handled by over loaded functions in the structure or property context, but also by means of property handle event handlers.

DB_undefined - Event is undefined

DBO_Initialize - Initializing an instance

Generated for: Instances

The event is generated when an instance has been initialized. This usually happens before reading or creating an instance or when using the GetInitInstance() property handle function.

The event handler may chnage attributes within the instance but cannot refer to referenced instances in collection properties.

DBO_Read - Read event

Generated for: Instances, Properties

The read event is generated when an instance has been read into the instance area of a property handle. When the event is generated, all properties are avaialble and can be accessed. The event handler allows e.g. filling transient properties or locating instances in subordinated collection properties.

DBO_Stored - Instance stored

Generated for: Instances, Properties

The event is generated after storing an instance to the database (or transaction). This happens, when the instance selection for a property handle has been changed and the last selected instance has been updated or when the instance is explicitely saved (Save()). The event handler allows e.g. re-filling transient properties or updating related instances.

The Instance stored must not be modified when handling this event. Modifying key components of the instance within this handler will cause database inconsistency.

DBO_Inserted - Instance inserted

Generated for: Collection Properties

The event is generated after an instance has been successfully inserted to a collection. It does not matter, whether the instance has been created or an existing instance has been added to a collection.

When the event is generated the instance is selected for the property handle, i.e. attributes and collection properties for the instance can be accessed.

DBO_Removed - Instance removed from collection

Generated for: Collection Properties

The event is generated after an instance has been successfully removed from a collection. It does not matter, whether the instance has been deleted or only removed from the collection.

When the event is generated the instance is not selected for the property handle, i.e. attributes and collection properties for the instance cannot be accessed.

DBO_Deleted - Instance deleted

Generated for: Instances

The event is generated after an instance has been successfully deleted. Usually, the deleted event is preceeded by a removed event (DBO_Removed).

When the event is generated the instance is not selected for the property handle, i.e. attributes and collection properties for the instance cannot be accessed.

DBP_Modify - Before Modify Instance

Generated for: Instances, Properties

The event is generated before updating the instance (e.g. when assining a different value to a property handle or when calling the Modify() function explicitly). When the event is generated the instance is selected and all attributes and collection properties can be accessed.

The function allows checking wether the modification is allowed and may reject the request if not.

DBP_Insert - Before Insert Instance

Generated for: Collection Properties

The event is generated before an instance is inserted to a collection. When the event is generated the selection state of the property handle depends on the application, i.e when an instance was selected in the property handle this is still available. Thus, values for initialising a new instance can be copied from the last instance selected. The new instance is not available at this time.

Usually, the insert event is followed by an initialize (and create - for new inszances) event. Only in case of a move operation the instance is selected when this event is generated and no initialize event is generated.

The event handler may check, wether insertion is allowed or not and may abort the request by returning an error (YES).

DBP_Remove - Before Remove Instance

Generated for: Collection Properties

The event is generated before an instance is removed from a collection. When the event is generated the instance is selected in the property handle and still accessible. Thus, the permission for removing the instance can be checked or other actions can be performed.

The remove event is followed by a delete event, when the action is executed for an owning or dependent collection.

The event handler may return an error (YES) for cancelling the request.

DBP_Delete - Before Delete Instance

Generated for: Instances

The event is generated before an instance is deleted. When the event is generated the instance is selected in the property handle and still accessible. Thus, the permission for deleting the instance can be checked or other actions can be performed.

The event handler may return an error (YES) for cancelling the request.

DBO_Opened - Instance or property opened

Generated for: Instances, Properties

The event is generated, when in instance or property handle has been opened. It allows initial settings in the context class for the instance or property.

DBO_Close - Property or instance context closed

Generated for: Instances, Properties

The event is generated, when in instance or property handle has been closed. It allows final actions in the context class for the instance or property. Close events cannot be denied.

DBP_Create - Create Instance

Generated for: Instances

The event is generated when crating a new instance. The event is not generated for imbedded structures but only for instances in references, relationships and shared base structures. The event is generated after the initialise event and allows initial settings for attributes in the instance. Subsequent collection properties cannot be accessed.

Creating an instance can be denied (e.g. no sufficiant access rights) by returning an error (YES) from the event handler.

DBO_Created - Instance created

Generated for: Instances

The event is generated sfter crating a new instance. The event is not generated for imbedded structures but only for instances in references, relationships and shared base structures. The event allows initial settings for attributes in the instance. Subsequent collection properties cannot be accessed.

Creating an instance can be denied (e.g. no sufficiant access rights) by returning an error (YES) from the event handler.

DBP_Store - Store Instance

Generated for: Instances, Properties

The event is generated before storing an instance to the database (or transaction) or a property (attribute) to an instance. The event is generated also for all imbedded structures and base structures. This happens, when the instance selection for a property handle has been changed and the last selected instance has been updated, when the instance is explicitely saved (Save()) or when assigning a new value to a property handle. The event handler allows e.g. re-filling transient properties or updating related instances.

The Instance stored must not be modified when handling this event. Modifying key components of the instance within this handler will cause database inconsistency.

Storing the instance can be denied by returning an error (YES) from the event handler.

DBO_NotCreated - No instance created

Generated for: Instances

Creating a new instance has terminated with a system error or by the DBP_Create event. More information about the error you may get from SDBError().

DBO_NotInserted - Instance not inserted

Generated for: Instances

Inserting an instance has terminated with a system error or by the DBP_Insert event. More information about the error you may get from SDBError().

DBO_NotOpened - Context not opened

Generated for: Instances, Properties

Opening an instance or property context has terminated with a system error or by the DBP_Open event. The property handle is not accessible. More information about the error you may get from SDBError().

DBO_NotRemoved - Instance not removed

Generated for: Instances

Removing an instance has terminated with a system error or by the DBP_Remove event. More information about the error you may get from SDBError().

DBO_NotDeleted - Instance not deleted

Generated for: Instances

Deleting an instance has terminated with a system error or by the DBP_Delete event. More information about the error you may get from SDBError().

DBO_Refresh - Refresh Event

Generated for: Properties

The refresh event indicates that the environment of a property handle has been changed. This usually happens for the sub property handles when another instance is selected in a parent property handle. In contrast to the read event the refresh event is generated only when the property handle is used, i.e. when being accessed or when an event handler is registered for the property handle. Thus, especially GUI applications are able to react imediately on changing collections.

The refresh event is used to update transient references or collections when an instance has changed. To avoid unecessary updates the refresh is generated only when data is requested from the property handle the first time after the parent handle has changed and not when reading the parent instance.

It is also possible to generate the event from the application using the property handle function Refresh().

DBP_Open - Opening instance or property context

Generated for: Instances, Properties

The event is generated, when an instance or property context is going to be opened. The property handle is accessible at this time, i.e. the event handler may access the instance collection (e.g. setting sort order or selecting an instance in a property handle).

Opening the context can be denied (e.g. no sufficiant access rights) by returning an error (YES) from the event handler.

DBP_Read - Before Read Event

Generated for: Instances

This event is generated for structure instances before reading an instance. At the time, when the event is generated, the instance is already selected in the property handle. Key data for the sort key is already available and can be copied to the instance area using the SetKey() structure context function or can be provided by using the GetKey() context function.

The handler can be used to optimize read access by returning an error or marking an instance as 'hidden' (HideInstance()), when an instance with the given key should not be provided.

DBP_Select - Select Instance

Generated for: Instances

An instance is going to be selected for the property handle. The new instance has already been selected for the property handle and attributes and collection properties are accessible.

The event handler may return an error (YES) to refuse the current selection of the instance, in which case the instance is unselected immediately.

DBP_Unselect - Unselect instance

Generated for: Instances

An instance is going to be unselected for the property handle. The instance is still selected and not stored (when being updated) and attributes and collection properties are accessible.

The event handler may return an error (YES) to refuse the unselection of the instance, in which case the instance remains selected.

DataSourceHandle - Data source

A data source describes an ODABA data source on a certain level (Dictionary, Database, DBObject, Extent, Instance). A data source can be parametrized by means of an INI-file. The INI file contains the names for the objects on the different levels. Not specified lower levels are not opened and have to be opened in the application (e.g. when defining only dictionary and database the extent is not opened and no instance is selected), The datasource is defined as section in the INI-file starting with the [datasource name].

A data source can be directed to a server. In this case the datasource has to be opened with a connected ODABA client or the INI-file must contain a server specification. In the last case the data source connects to the server automatically when opening the data source. The connection is owned by the datasource in this case.

A data source cane be opened and closed as a whole (Open(), Close()) or separately on each definition level (Connect(), OpenDictionary(), ...).

BeginTransaction - Start transaktion for the data source

Data sources provide simple transaction control. Data source transactions cannot be nested, i.e. when a transaction is running for the datasource no other transaction can be started.

Using nested transactions is possible with the DBObjectHandle.

```
logical DataSourceHandle :: BeginTransaction (logical ext_ta )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
ext_ta	External Transaction The option indicates that updates made within the transaction should be stored on a disk. This option should be set when the transaction is a long transaction that helds many (100 000) or more updates in the transaction.

Close - Close DataSourceHandle

The handles on all hierarchy levels (Dictionary to Extent) are closed when they are owned (opened) by the Data-SourceHandle.

```
logical DataSourceHandle :: Close ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

CloseDBObject - Close DBObjectHandle

The DBObjectHandle and subsequent handles (PropertyHandles) are closed beginning with the lowest opened handle. Handles are closed only when they have been opened by the datasource handle. Property handles opened by the application must be closed by the application before.

```
logical DataSourceHandle :: CloseDBObject ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

CloseDatabase - Close DatabaseHandle

The DatabaseHandle and subsequent handles (DBObjectHandle and PropertyHandle for Extents) are closed beginning with the lowest opened handle. Handles are closed only when they have been opened by the datasource handle.

When a data source transaction has is still running it will be commited.

```
logical DataSourceHandle :: CloseDatabase ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

CloseDictionary - Close DictionaryHandle

The DictionaryHandle and subsequent handles for Database, DatabaseObject and Extent are closed beginning with the lowest opened handle. Handles are closed only when they have been opened by the datasource handle.

```
logical DataSourceHandle :: CloseDictionary ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

CloseProperty - Close PropertyHandle

The PropertyHandle for the defined extent is closed. Handles are closed only when they have been opened by the datasource handle.

```
logical DataSourceHandle :: CloseProperty ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

CommitTransaction - Terminate transaction

The datasource transaction it stopped and modifications are stored to the database. Commiting the data source transaction will commit all subsequent DBObjectHandle transactions that are still running.

```
logical DataSourceHandle :: CommitTransaction ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Connect - Connect DataSourceHandle to server

The function connects the DataSourceHandle to a server. The server name (server_name) and port number (port_number) must be defined in the DataSourceHandle before calling this function.

If no client object (odaba_client) is passed a client object is created when a server is defined. If no server name is defined the function does not try to connect.

```
logical DataSourceHandle :: Connect (ODABAClient &odaba_client )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
odaba_client	ODABA Client Handle The ODABA client handle can be passes as connectet or ea empty handle.

DataSourceHandle - Construcktor

The constructor creates a DataSourceHandle. Before openeing a datasource database pathes and object and extent names have to be set. This can be done by explicitly setting the pathes and names in the programm or by means of an INI-file using the SetupVariables() function..

```
DataSourceHandle :: DataSourceHandle (
)
```

Disconnect - Disconnect from server

The function disconnects from the server. When the DataSourceHandle is still opened it is closed (Close()) before disconnecting.

Disconnecting will delete the ODABAClient when it has been created by the DataSourceHandle.

```
logical DataSourceHandle :: Disconnect ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

GetClient - Get client handle from data source

The function returns the client handle from the data source.

```
ODABAClient &DataSourceHandle :: GetClient ( )
```

Return value	The ODABA client handle can be passes as connectet or ea empty handle.

Open - Open DataSourceHandle

A Datasource consists of a Dictionary and a Database. In addition a DatabaseObject and an Extent can be defined. The data source can be closed using the function Close(). If a DataSourceHandle is already opened this is closed before reopening the DataSourceHandle with the current parameters.

The data source is defined by means of external resource defintions in the DataSourceHandle (server_name, dict_path, db_path, object_name, extent_name, inst_key).

i0 - Open empty data source

A data source can be opened when the application has filled the external specifications for the data source. At least the dict_path should be set in the data source before openeing.

The data source should be opened only, when all external resources to be accessed are defined in the data source. When the client is not connected the server specifications (server_name and port_number) shuold be set as well. This is not necessary when running a local application.

```
logical DataSourceHandle :: Open (ODABAClient &odaba_client,
                                  PIACC acc_mod )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
odaba_client	ODABA Client Handle The ODABA client handle can be passes as connectet or ea empty handle.

acc_mod	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write). Default: PI_Read

i02 - Opening datasource based on ini-file section

The function opens the data source based on external resource definitions passsed in the ini-file. The inifile section name containing the resource definitions is passed as datasource_name to the function. The resource definitions in the data source are filled from the corresponding resource definitions in the ini-file.

```
logical DataSourceHandle :: Open (ODABAClient &odaba_client,
                       char *ini_file, char *datasource_name )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
odaba_client	ODABA Client Handle The ODABA client handle can be passes as connectet or ea empty handle.
ini_file	Application ini-file The ini-file may contain several sections providing application or data source information. The path to the ini-file is passed as 0-terminated string.
datasource_name	Name of the data source The name of the data source defines the section in the INI-file or an entry in the data catalogue that contains the external data source definitions.

i1 - Opening data source from catalogue

The function opens the data source based on external resource definitions passsed stored in the data source catalogue. In this case the external resource defitions are read from the catalogue where an entry with the passed datasource_name must exist.

Using the catalogue feature the catalogue data source must be provided in a catalogue [DATA-CATALOGUE] section of the ini-file (local application) or in a corresponding section of the server. The resource definitions in the data source are filled from the corresponding resource definitions in the catalogue entry before opening the data source.

```
logical DataSourceHandle :: Open (ODABAClient &odaba_client,
                             char *datasource_name )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
odaba_client	ODABA Client Handle The ODABA client handle can be passes as connectet or ea empty handle.
datasource_name	Name of the data source The name of the data source defines the section in the INI-file or an entry in the data catalogue that contains the external data source definitions.

OpenDBObject - Open DBObjectHandle

The DBObjectHandle is opened for the database object defined in the database path (object_name). The DataBaseHandle of the DataSource must be opened before. If the DBObjectHandle is already opened it will be closed before opening the new DBObjectHandle (CloseDBObject()).

When the object name is empty the root object of the opened database is provided.

```
DBObjectHandle &DataSourceHandle :: OpenDBObject ( )
```

Return value

OpenDatabase - Open DatabaseHandle

The DatabaseHandle is opened for the database defined in the database path (db_path). The DictionaryHandle of the DataSource must be opened before. If the DatabaseHandle is already opened it will be closed before opening the new DatabaseHandle. (CloseDatabase()).

When the database path is empty the opened DictionaryHandle is opened as DatabaseHandle.

```
DatabaseHandle &DataSourceHandle :: OpenDatabase ( )
```

Return value	This is pointer to an opened database handle. The database handle can be an opened database handle DatabaseHandle as well as an opened dictionary handle (DictionaryHandle).

OpenDictionary - Open DictionaryHandle

The DictionaryHandle is opened only when a dictionary path has been defined (dict_path). If another dictionary has already been opened it is closed (CloseDictionary()) before re-opening the DictionaryHandle. The DictionaryHandle is opened with the access mode passed to the function. When no database path (db_path) is defined and no access mode is passed the DictionaryHandle is opened with the access mode defined for the database,

```
DictionaryHandle &DataSourceHandle :: OpenDictionary (PIACC ac-
                 copt )
```

Return value	An opened dictionary handle is passed.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).

OpenProperty - Open PropertyHandle

The PropertyHandle can be opened only when the DBObjectHandle is opened for the data source and an extent name (extent_name) has been specified. If a PropertyHandle is already opened it will be closed (CloseProperty()) before re-open the handle.

```
PropertyHandle *DataSourceHandle :: OpenProperty (char *extname
                 )
```

Return value	Is a pointer to an (usually) opened property handle.
extname	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

RollBack - Roll back modifications made in the transaction

All modification made since the transaction has been started are removed. The transaction is stopped. If there sur subsequent transactions opened by DBObjectHandles those are closed as well.

```
logical DataSourceHandle :: RollBack ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

SetDBObject - Set DBObjectHandle

The function allows setting an opened DBObjectHandle as DBObjectHandle for the DataSourceHandle. A DBObjectHandle set with this function is not closed when calling (CloseDBObject()). When the DataSourceHandle has already opened it will be closed (CloseDBObject()) before setting the new DBObjectHandle.

```
logical DataSourceHandle :: SetDBObject (DBObjectHandle
                   &ohandle, char *w_objname )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
ohandle	

SetDataSource - Set data source definitions

The function sets the external and internal resources as copied from the passed data source handle. Opened internal resources (access handle) will not be closed when closing or destructing the data source handle but when closing the original data source handle.

```
logical DataSourceHandle :: SetDataSource (DataSourceHandle
                     *dbdefptr )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbdefptr	Data source handle The data source handle contains definitions for external and internal resources (resource names and opened resource handles)

SetDatabase - Set DatabaseHandle

The function allows setting an opened DatabaseHandle as DatabaseHandle for the DataSourceHandle. When the DataSourceHandle has already an opened DatabaseHandle this is closed (CloseDatabase()) before setting the handle passed to the function.

A DatabaseHandle set with this function will not be closed when closing the data source or the database handle (Close(), CloseDatabase()).

```
logical DataSourceHandle :: SetDatabase (DatabaseHandle
                     &db_handle, char *w_basepath )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
db_handle	Pointer to database handle This is pointer to an opened database handle. The database handle can be an opened database handle DatabaseHandle as well as an opened dictionary handle (DictionaryHandle).
w_basepath	Path for the opened database handle The path for the opened database handle can be passed to set the original database path in the data source.

SetDictionary - Set DictionaryHandle

The function allows setting an opened DictionaryHandle as DictionaryHandle for the DataSourceHandle. A DictionaryHandle set with this function is not closed when calling (CloseDictionary()). If the DataSourceHandle has already an opened DictionaryHandle it will be closed (CloseDictionary()) before setting the new handle.

```
logical DataSourceHandle :: SetDictionary (Dictionary *dictptr,
                          char *w_dictpath )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dictptr	Dictionary handle An opened dictionary handle is passed.

SetVariables - Set variables from INI-File

The function Initializes the DataSourceHandle from the data source section in the INI-file.

The data source variables must be defined in a section [datasource_name] in the INI-file. This section may contain the following variables:

ODABA_SERVER=server_name

The server_location specifies the server name (e.g. ODABAServer1). When no server is specified the resources are supposed to be available locally.

ODABA_SERVER_PORT=server_port

The port number must be the same that has been used for starting the servers (default is 6123).

DICTIONARY=dict_path

This variable defines the resource database (dictionary). This variable is mandatory. The value may refer to a server variable that defines the path on the server. Server variables must be enclosed in % characters (e.g. %DICT_PATH%).

DATABASE=db_path

This variable defines the complete path to the database that contains the application data. The value may refer to a server variable that defines the path on the server. Server variables must be enclosed in % characters (e.g. %DB_PATH%).

WORKSPACE=workspace

When the workspace feature is enabled for the database a workspace can be defined as active workspace for the data source by passing a workspace name or a workspace path..

OBJECT=object_name

The name of database object must be specified if a sub object space in the database is to be opened..

EXTENT=extent_name

Name of an extent when the DataSource refers to a certain collection.

STRUCTURE=struct_name

The structure name is used in some cases for performing metadata operetions (e.g. copying a structure definition to another dictionary). It is has no direct influence on the data source but can be retrieved from the application.

SCHEMA_VERSION=schema_version

Schema version when the application should be opened

```
logical DataSourceHandle :: SetVariables (char *datasource_name
                    )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
datasource_name	Name of the data source The name of the data source defines the section in the INI-file or an entry in the data catalogue that contains the external data source definitions.

Setup - Setup data source parameters

The function updates the external resource references from the ini-file and/or the data catalogue. The ini-file passed to the function is set as current ini-file for the data source.

External definitions are copied from the ini-file or data catalogue section according to the passed datasource_name into the data source.

```
logical DataSourceHandle :: Setup (char *ini_file, char
                    *datasource_name )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
ini_file	Application ini-file The ini-file may contain several sections providing application or data source information. The path to the ini-file is passed as 0-terminated string.
datasource_name	Name of the data source The name of the data source defines the section in the INI-file or an entry in the data catalogue that contains the external data source definitions.

SetupVariables - Setup data source variables from INI-file

The function tries to initialize the data source parameters from a section defined in the ini-file. If no such section is defined or the section refers to a data source in the catalogue the function tries to setup the variables from the corresponding catalogue entry.

```
logical DataSourceHandle :: SetupVariables (char
                 *datasource_name )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
datasource_name	Name of the data source The name of the data source defines the section in the INI-file or an entry in the data catalogue that contains the external data source definitions.

~DataSourceHandle - Destructor

Destructing the datasource will close all handles that are not yet closed. When a data source transaction is still running this will be comitted before destroying the data source.

When working with recovery file thsi is closed as well.

```
                        DataSourceHandle :: ~DataSourceHandle (
                 )
```

DataTypes - ODABA data types

CHAR - Character

Character fields are defined with a maximum number of characters. The maximum number of characters in a character field is limited to 2 GB.

CCHAR - Coded character

Coded character fields are used when data stored in the database must be encrypted (e.g. when storing pass-words in the database). The number of characters in a coded character field is limited to 256 characters.

STRING - String character

String fields are character fields with variable size. The end of the string is indicated with a 0-character (0-terminated string). 0-termination will only inluence the representation of the value. String fields are always stored with the full defined size in the database.

MEMO - Memo character

Memo character fields are considered as 0-terminated text fields (like STRING). Memo character fields are used to store large text fields in the database. Since memo fields are not directly stored in the instance they will occupy storage only in the used size.

INT - Signed integer or decimal number

ODABA consideres decimal velues the same way as integer values by defining a precision. The size fir an integer value defines the number of significant digits.

REAL - Float point number

REAL allows defining float point numbers. For REAL fields the size defines the number of digits for representing the field. Values upto 8 define a 4-byte float point number, values upto 17 define a 8 byte float point number.

LOGICAL - Logical field

Logical fields may only contain bool values true (YES) and false (NO). The size for a logical field should always be 1.

DATE - Date

Date fields allow storing date values. The size for a date filed influences only the standard presentation of the date, i.e. the date-to-string conversion. Following size definitions are possible:

8:	"2002/09/14"
6,7:	"02/09/14"
4,5:	"02/09"
2,3:	"02"

Independent on the size the date is stored always with 4 byte.

TIME - Time

Time fields allow storing time values from 0:00:00,00 upto 23:59:59,99. The size for a time filed influences only the standard presentation of the time, i.e. the time-to-string conversion. Following size definitions are possible:

8:	"23:59:59,99"
6,7:	"23:59:59"
4,5:	"23:59"
2,3:	"23"

Independent on the size the time is stored always with 4 byte.

DATETIME - Timestamp

A date/time field is considered as basic data type even thoug it seems structured like date and time. The size for the date time field will influence neither the storage size nor the presentation (to-string conversion). Timestamps are always presented as

"2002/09/14|23:59:59:99"

VOID - Unknown type

Unknown types can be defined for references only. VOID references may refer to a collection or a single object instance with unknown type. The database will determine the instance type at run-time in this case.

BIT -

DatabaseHandle - Database Handle

Database handle must be created for accessing data in an ODABA database. An ODABA database must be connected with a dictionary, which defines the object model for the database.

Each ODABA database consists of at least one Database Object (Root Object) that is the owner od extents and other data collections.

When creating a database handle the object handle this is based on a database object handle (-> DBObjectHandle) for the root object, i.e. the database handle inherits all the functionality from the database object handle.

A database may consists of a number of physical separated mainbases, sub-bases and data areas. This is, however, handles internally after creating the database. For creating a multiple resource database the database handle provides several functions for initializing main and sub bases and data areas.

Moreover, the database handle provides log and recovery features, that allow logging all changes made on the database or recovering the database in case of errors.

The workspace feature supported by the database handle is a sort of persistent transactions. It allows storing changes for a longer period outside the database and consolidating or discarding changes when requested by the user.

ActivateShadowBase - Activate Shadow Database

When running a shadow database (e.g. when worspace support is enabled) you might want to read information from the shadow database rather than from the original database. Since the shadow database contains the information including all updates made in workspeces and not yet published, the shadow database is the only place where logical consistency checks can be made.

The function switches from the original database to the shadow database and allows reading from the shadow database, i.e. all read operations are directed to the shadow database instead of the original database. This funktion has no effect when the shadow database feature is not enabled.

When the shadow database is activated restricted updates are possible on instances (you may not change properties that are referenced as key components).

After performing the checking or other tasks you must deactivate the shadow database (DeactivateShadowBase()).

```
logical DatabaseHandle :: ActivateShadowBase ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ChangeRecovery - Enable/disable recovery support

This function allows you to enable or disable the recovery support for the opened database.

To disable the recovery support you can pass 'RECOVERY_none' as recovery type.

To enable recobery support you can pass 'RECOVERY_full' or 'RECOVERY_transaction'. You should enable the recovery support immediately after the latest backup.

```
logical DatabaseHandle :: ChangeRecovery (RecoveryType rec_type,
                          char *rec_path )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
rec_type	Type of recovery support
rec_path	Folder for recovery files

CheckLicence - Check Application License

The function checks the license number for the user. The function returns an error (YES) when the database or the application is not licensed.

i01 - Register costumer

The function checks the user name and license number and registers the license information in the database. When licensing is requested the license information is checked whenever the database is opened.

```
logical DatabaseHandle :: CheckLicence (char *lic_owner, char
                           *lic_number )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
lic_owner	Owner of the licence
lic_number	Licence number The licence number consists of twelve alphanumeric characters.

i02 - Check application licence

The function checks the license for the application name passed to the function. Applications may request specific licenses, which can be checked with this function.

```
logical DatabaseHandle :: CheckLicence (char *applname_w )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

applname_w

CloseRecovery - Close recovery file

The function closes the recovery file. Usually the recovery file is closed when closing the database and should not be closed explicitely ba the application.

```
logical DatabaseHandle :: CloseRecovery ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

CloseWorkspace - Close Workspace

Workspaces are closed when closing the database. It is possible, however, to close the active workspace explicitly.Closing the workspace will not save the changes in the database or lower workspace but keep until the workspace is opened again.

```
logical DatabaseHandle :: CloseWorkspace ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ConsolidateWorkspace - Consolidate Workspace

The function will consolidate all changes made in the workspace. You can consolidate the currently opened workspace, only, i.e. you must open the workspace before consolidating. For consolidating a workspace it must be opened with exclusive use. Only when no other user has access to the workspace it is possible to consolidate it.

```
logical DatabaseHandle :: ConsolidateWorkspace ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DatabaseHandle - Konstructor

A database can be opened in local, in client/server mode or in file server mode. Local mode usually implies exclusive access. When running several applications on a local machine the database should be opened in file servermode to provide concurrent access to the database. Client/server mode is suggested when running the database from different clients on a central server.

a1 - Create database handle

The function creates a database handle for an opened dictionary. The database path (cpath) passed to the constructor may contain system variable references that are resolved before opening the database.

The database can be opened in read or write mode (accopt). When running the database in file server mode the netoption defines whether the database is running exclusive or can be shared by other users. The local_resources parameter defines the way the database is opened.

For opening an older version for the database you may pass a version number in version_nr.

```
     DatabaseHandle :: DatabaseHandle (Dic-
tionaryHandle &dict_handle, char *cpath, PIACC
accopt, logical w_netopt, logical
online_version, uint16 version_nr, Resource-
Types local_ressources, char sysenv )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dict_handle	Dictionary handle The dictionary handle usually refers to an opened dictionary. To check whether a dictionary is opened you can use the !-operator.
cpath	Complete path The complete path is passed as 0-terminated string with a maximum length of 255 characters.
accopt	Access option

	The access option defines the way instances in a property handle are to be accessed (read, update, write).
w_netopt	Multi-user option YES indicates that multi-user access is requested. NO indicates exclusive use of database. Accessing a database in update or write mode, NO guarantees absolute exclusive access.
online_version	Online versioning option When this option is set the database will be enabled vor online versioning. When the option is set to NO the system variable ONLINE_VERSION is checked instead. Default: NO
version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION
local_ressources	Resource type Depending on the resource type the database or dictionary is opened on the client or server side. **RES_automatic** When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side. **RES_local** The dictionary or database will be opened on the client machine in any case. **RES_server** The dictionary or database will be opened on the server machine side in any case.
sysenv	System application

This option indicates that the application is running as system application. In this case context functions are disabled and will not be executed. This option should never be set in normal applications because this may lead to logical inconsistence of the database.

c1 - Copy constructor

This constructor creates a copy of the database handle. Both, the copy and the origin are referring to the same resources. The database handle is closed when closing the last database handle instance for a database, regardles on the sequence the handles have been opened.

```
                    DatabaseHandle :: DatabaseHandle (const
DatabaseHandle &dbhandle_refc )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbhandle_ref	

c2 - Create a copy of the handle

This constructor creates a copy of the database handle. Both, the copy and the origin are referring to the same resources. The database handle is closed when closing the last database handle instance for a database, regardles on the sequence the handles have been opened.

```
                    DatabaseHandle :: DatabaseHandle (DBHan-
dle *_dbhandle )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
_dbhandle	

d1 - Create database

This constructor allows creating a new database. Usually creating a database explicitly is not necessary. When, however, special options as low est and higest local identifiers (LOID) are to be passed this constructor can be used.

```
        DatabaseHandle :: DatabaseHandle (char
*cpath, int16 lowEBN, int16 highEBN, int32
dasize, logical largedb, logical pindep )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
cpath	Complete path The complete path is passed as 0-terminated string with a maximum length of 255 characters.
lowEBN	First entry number in database Low range number for the mainbase. Depending on the database (small or large) the range number is between 0 and 252 (small DB) or 0 and 32767 (0x7fff) for large databases.
highEBN	Last entry number in database High range number for the mainbase. Depending on the database type the range number is between 0 and 252 (small DB) or 0 and 32767 (0x7fff) for large databases.
dasize	Size for data area The data area size allows limiting the area for the data area. When no data area is passed (UNDEF), the data area expands whenever more space is needed.
largedb	Large database option The large database option idicates that a large database is to be defined. This information is stored in the database header after creating the database.
pindep	Platform independance option The plattform independance option idicates that integer numbers are to be stored in platform independent format. This information is stored in the database header after creating the database.

d2 - Create database handle for dictionary

The function creates a database handle from the dictionary handle.

```
                    DatabaseHandle :: DatabaseHandle (Dic-
tionaryHandle &dict_handle, PIACC accopt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dict_handle	Dictionary handle The dictionary handle usually refers to an opened dictionary. To check whether a dictionary is opened you can use the !-operator.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).

x1 - Empty database handle

The constructor creates an empty database handle. A database can be opened later with this handle using the Open() function.

```
                    DatabaseHandle :: DatabaseHandle ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DeactivateShadowBase - Deactivate Shadow Database

Deactivating the shadow database causes all read operations being sent to the original database again. This funktion has no effect when the shadow database feature is not enabled.

```
logical DatabaseHandle :: DeactivateShadowBase ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DeleteWorkspace - Delete workspace

The function deletes an existing workspace. The workspace must be empty before deleting, i.e. discard or consolidate must run before.

```
logical DatabaseHandle :: DeleteWorkspace (char *ws_names, char
                          *user_name )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
ws_names	Workspace name The workspace name is the extension of the current workspace or database. The database can be considered as the root for all workspaces. The workspace name may address a workspace on top of the current one (simple workspace name) or a workspace on any higher level by passing a sequence of workspace names separated by '.'.
user_name	User name When accessing user protected resources as databases or workspaces, a user must be passed as 0-terminated string, otherwise NULL.

DisableWorkspace - Disabeling workspace feature

Disabeling the workspace feature requires that all workspaces have been discarded or consolidated. If this is not the case active workspaces must be consolidated before.

When disabling the workspace feature sucsessfully the shadow database is removed as well.

```
logical DatabaseHandle :: DisableWorkspace ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

DiscardWorkspace - Discard Workspace

The function will through away all changes made in the workspace for the currently opened workspace. The workspace will be closed and removed.

```
logical DatabaseHandle :: DiscardWorkspace ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

EnableWorkspace - Enable workspace feature

The function enables the usage of workspaces and shadow database. The function can be executed only when the database is opened exclusive.

When no path or an empty path is passed as location for the shadow database the shadow database is positioned in the same folder as the original database and with the same name as the database but the extension is changed to .sdw or appended if no extension has been defined for the database.

```
logical DatabaseHandle :: EnableWorkspace (char *sdw_path )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
sdw_path	Complete path for shadow database The path for the shadow database is passed as 0-terminated string with a maximum size of 255.

ExecuteDatabaseAction - Execute action on database level

The function calls an action that is defined in the database context. The function is executed on the server side first. If it was executed successfully, the function is executed on the client side, too.

The action may use the SetActionResult() function to pass the result of the action to the client application. If execution of the function on the client side returns NO the result passed from the server overwrites any result set by the client function. The result can be retrieved from the client application using the function GetActionResult().

```
logical DatabaseHandle :: ExecuteDatabaseAction (char
                 *action_name, char *parm_string )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
action_name	Name of the action to be performed The name of the action is passed as 0-terminated string with a maximum length of 40 significant characters.
parm_string	Parameter string The parameter string is passed as 0-terminated string and contains the parameters according to the conventions of the action called.

ExistWorkspace - Exist workspace?

The function returns whether a workspace with the passed workspace name (ws_name) exists as subordinated workspace (YES) or not(NO). When the database has already opened a workspace the function looks for the workspace relatively to the opened one.

```
logical DatabaseHandle :: ExistWorkspace (char *ws_names )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
ws_names	Workspace name The workspace name is the extension of the current workspace or database. The database can be considered as the root for all workspaces. The workspace name may address a workspace on top of the current one (simple workspace name) or a workspace on any higher level by passing a sequence of workspace names separated by '.'.

GetDatabaseID - Get database resource number

The function returns the database resource number.

```
int32 DatabaseHandle :: GetDatabaseID ( )
```

Return value

The database resource number is a number that has been assigned to the database in the database definition of the dictionary. The database number does not describe a database as such but more a type of data bases. It is used to assign the database context class, which is associated with the database resource number.

GetPath - Get path for the opened database

The function returns the database path for the opened database. The path returned refers to the database path and does not contain system variable references anymore.

```
char *DatabaseHandle :: GetPath ( )
```

Return value

The complete database path is passed as 0-terminated string with a maximum length of 255 characters.

GetRecoveryFile - Provide name of recovery file

The function returns the name of the recovery file according to the passed recovery number.

```
char *DatabaseHandle :: GetRecoveryFile (uint16 recnum )
```

Return value

The name of the recovery file is passed as 0-terminated string. The recovery file name has been generated when creating the recovery file ({.r DatabaseHandle.InitRecovery}()).

recnum

Numer of recovery file

Recovery files have an internal number that is generated when creating the recovery file ({.r DatabaseHandle.InitRecovery}()).

GetRecoveryNum - Provide reacovery number

The function returns the number for the current recovery file.

```
uint16 DatabaseHandle :: GetRecoveryNum ( )
```

Return value

Recovery files have an internal number that is generated when creating the recovery file ({.r DatabaseHandle.InitRecovery}()).

GetRecoveryPath - Provide path for recovery folder

The function returns the path for the folder containing the recovery files.

```
char *DatabaseHandle :: GetRecoveryPath ( )
```

Return value	The recovery path points to a folder that contains the recovery files. The folder path is passed as 0-terminated string. The folder has been defined when creating the recovery file. ({.r DatabaseHandle.InitRecovery}()).

GetSchemaVersion - Get schema version

The function returns the current schema version number for the database.

```
uint16 DatabaseHandle :: GetSchemaVersion ( )
```

Return value

GetSystemVersion - Get system version

The fiunction provides the schema version of the ODA-BA system, which is the dictionary for a dictionary.

```
uint16 DatabaseHandle :: GetSystemVersion ( )
```

Return value

GetVersionString - Provide database version

The function provides the database version and sub version as string.

```
char *DatabaseHandle :: GetVersionString ( )
```

Return value	The version string is passed as 0-terminated string like e.g. "2.41".

GetWorkspace - Get workspace names

The function returns the workspaces defined below an existing workspace or database. The function returns workspace names by index sorted in alphabetical order (first entry is retrieved with index 0). Only workspaces on a given level are returned. To get workspaces on lower levels you must pass the root path for the lower level.

Workspace information is buffered when retrieving it the first time. To refresh the internal workspace list you should pass the refresh option (YES).

For retrieving workspaces owned by the user a user name can be passed. Not passing a username will return all workspaces.

The name for the workspace is returned in ws_name in addition when passing a pointer to a character array. Otherwise the name is returned only in the result area of the property handle, which might be destroyed after the next property handle function call.

```
char *DatabaseHandle :: GetWorkspace (char *ws_root, int32
                   ws_index, char *user_name, char *ws_name, log-
                   ical refresh_opt, char *ws_info )
```

Return value	The workspace name is a 0-terminated string with a maximum size of 128 characters, which contains the name of the workspace without the preceeding workspace path for the hierarchy of upper workspaces.
ws_root	Workspace root The workspace root is a 0-terminated string that describes the hirarchy of upper workspaces. The hierarchy is described by workspace names separated by '.'.
ws_index	Number of workspace to be retrieved This is the internal number of workspace to be retrieved. The first workspace is retrieved by index 0.
user_name	User name When accessing user protected resources as databases or workspaces, a user must be passed as 0-terminated string, otherwise NULL.
ws_name	Work space name

The workspace name is a 0-terminated string with a maximum size of 128 characters, which contains the name of the workspace without the preceeding work-space path for the hierarchy of upper workspaces.

refresh_opt	Refresh option Setting the refresh option to YES will rebuild the list completely.
ws_info	Workspace information A character array with minimum size of 256 characters can be passed that carries additional workspace infor-mation when the function has terminated successfully. Additional nformation is passed as 0-terminated string: \i ws_name (ID=ws_number[; Us-er=user_name])

IgnoreWriteProtect - Ignor permanent write protection

The function allows disabling the permanent write pro-tection. After disabling permanent write protection in-stances that have been marked as permanent write pro-tected (-> SetWProtect()) can be updated for this data-base handle.

```
logical DatabaseHandle :: IgnoreWriteProtect ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are availa-ble in the error object.

InitDataArea - Initialise DataArea

The function allows initializing a new data area. Data areas must be created consecutive order. A data area 0 is created automatically, when creating the upper sub-base, i.e. the next data area to be crreated would be data area 1 etc.

```
logical DatabaseHandle :: InitDataArea (int16 mbnumber, int16
                          sbnumber, int16 danumber, char *filename,
                          int32 dasize )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are availa-ble in the error object.

mbnumber	Main base number Mainbase numbers from 0 to 252 (for small databases) and 0 to 32767 (for large databases) are valid.
sbnumber	Sub-base number Sub-bases for a main base are numbered contineously. The highest sub-base number is 255.
danumber	Data area number Data areas for a sub-base are numbered contineously. The highest data area number is 255.
filename	File name for DataArea file The file name is passed as 0-terminated string with a maximum length of 80 characters. The path may contain symbolic parameters, which are replaced by the value of a corresponding system variable or variable set in the INI-file (e.g.in case of "%ROOT%\base1.rot" - %ROOT% is replaced by the value of the ROOT system or INI file variable). The replacement is done only for utility applications, i.e. a utility control block must have been created (see **{.r UtilityCB**}).
dasize	Size for data area The data area size allows limiting the area for the data area. When no data area is passed (UNDEF), the data area expands whenever more space is needed.

InitMainBase - Initialize main base

The function allows initializing a new main base. Main bases must be created in consecutive order. The first main base to be crreated would be main base 0, the next main base 1 etc. A main base 0 is created automatically, when creating a single resource database.

Creating a mainbase automatically creates a sub-base 0 and a data area 0. Data area size (dasize) and file name refer to data area 0.

Main bases are generating local identities. The size for local identities depends on the database type and is 64 bit for large databases and 32 bit for small databases:

large DB: 0xSSRRRRNNNNNNNNNN

small DB: 0xRRNNNNNN

'SS' is used internally for the system. RR or RRRR is the part of the identity that is described by the range of identities for the main base, i.e. a mainbase generates identities with range values according to the low and high value passed (lowEBN, highEBN).

Usually a database has certain limitations to 2 or 4 giga byte (31 bit). This is sufficiant in many cases but some sutuations require more space. Allocating a large database (YES) will change the following limitations:

Data area size: 2 GB to 262144 GB

number of identities: ca 4 Giga to more than 16,000,000 Giga

number of mainbases: 252 to 32760

Large databases do not support, however, main bases splitted in different sub bases and data areas. This feature has been introduced to overcome database limitations for small databases and is not necessary for large databases.

Because of different interger presentations on different platforms databases are platform dependent and must be converted when changing the platform. It is, however, also possible to store data in platform independent format when passing YES forte pindep parameter.

```
logical DatabaseHandle :: InitMainBase (int16 mbnumber, char
                 *filename, int16 lowEBN, int16 highEBN, int32
                 dasize, logical largedb, logical pindep )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
mbnumber	Main base number Mainbase numbers from 0 to 252 (for small databases) and 0 to 32767 (for large databases) are valid.
filename	File name for DataArea file The file name is passed as 0-terminated string with a maximum length of 80 characters. The path may contain symbolic parameters, which are replaced by the value of a corresponding system variable or variable set in the INI-file (e.g.in case of "%ROOT%\base1.rot" - %ROOT% is replaced by the value of the ROOT system or INI file variable). The replacement is done only for utility applications, i.e. a utility control block must have been created (see **{.r UtilityCB**}).
lowEBN	First entry number in database Low range number for the mainbase. Depending on the database (small or large) the range number is between 0 and 252 (small DB) or 0 and 32767 (0x7fff) for large databases.
highEBN	Last entry number in database High range number for the mainbase. Depending on the database type the range number is between 0 and 252 (small DB) or 0 and 32767 (0x7fff) for large databases.
dasize	Size for data area The data area size allows limiting the area for the data area. When no data area is passed (UNDEF), the data area expands whenever more space is needed.
largedb	Large database option The large database option idicates that a large database is to be defined. This information is stored in the database header after creating the database.
pindep	Platform independance option The plattform independance option idicates that integer numbers are to be stored in platform independent format. This information is stored in the database header after creating the database.

InitRecovery - Initialise recovery file

The function initializes a new recovery file.

```
logical DatabaseHandle :: InitRecovery (char *recpath, uint16
                          recnum )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
recpath	Recovery path The recovery path points to a folder that contains the recovery files. The folder path is passed as 0-terminated string. The folder has been defined when creating the recovery file. ({.r DatabaseHandle.InitRecovery}()).
recnum	Numer of recovery file Recovery files have an internal number that is generated when creating the recovery file ({.r DatabaseHandle.InitRecovery}()).

InitSubBase - Initialise sub-base

The function allows initializing a new sub-base. Sub-bases must be created in consecutive order. A sub-base 0 is created automatically, when creating the upper main base, i.e. the next sub-base to be crreated would be sub-base 1 etc.

Data area size (dasize) and File name refer to data area 0, which is automatically allocated with the sub-base.

```
logical DatabaseHandle :: InitSubBase (int16 mbnumber, int16
                          sbnumber, char *filename, int32 dasize )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
mbnumber	Main base number Mainbase numbers from 0 to 252 (for small databases) and 0 to 32767 (for large databases) are valid.
sbnumber	Sub-base number Sub-bases for a main base are numbered contineously. The highest sub-base number is 255.

filename	File name for DataArea file The file name is passed as 0-terminated string with a maximum length of 80 characters. The path may contain symbolic parameters, which are replaced by the value of a corresponding system variable or variable set in the INI-file (e.g.in case of "%ROOT%\base1.rot" - %ROOT% is replaced by the value of the ROOT system or INI file variable). The replacement is done only for utility applications, i.e. a utility control block must have been created (see **{.r UtilityCB**}).
dasize	Size for data area The data area size allows limiting the area for the data area. When no data area is passed (UNDEF), the data area expands whenever more space is needed.

IsLicenced - Is database licensed

The function returns whether the database has been licensed successfully. Usually the database will not be opened when a license is required and the database is not licensed. When running with disabled license services this function can be used to check the license after opening the database.

```
logical DatabaseHandle :: IsLicenced ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsShared - Is database shared by several users

The function returns YES when the database has been opened in net mode on a local machine or when running in a client/server environment.

```
logical DatabaseHandle :: IsShared ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

LocateWorkspace - Locate an existing Workspace

The function checks whether the worspace with the passed name exists relatively to the current workspace.

```
logical DatabaseHandle :: LocateWorkspace (char *ws_names )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
ws_names	Workspace name The workspace name is the extension of the current workspace or database. The database can be considered as the root for all workspaces. The workspace name may address a workspace on top of the current one (simple workspace name) or a workspace on any higher level by passing a sequence of workspace names separated by '.'.

Open - Opening a database handle

The function allows opening a database handle. When the database handle is already opened it will be closed before re-opening it.

A database can be opened in local, in client/server mode or in file server mode. Local mode usually implies exclusive access. When running several applications on a local machine the database should be opened in file servermode to provide concurrent access to the database. Client/server mode is suggested when running the database from different clients on a central server.

a1 - Open database handle

The function creates a database handle for an opened dictionary. The database path (cpath) passed to the function may contain system variable references that are resolved before opening the database.

The database can be opened in read or write mode (accopt). When running the database in file server mode the netoption defines whether the database is running exclusive or can be shared by other users. The local_resources parameter defines the way the database is opened.

For opening an older version for the database you may pass a version number in version_nr.

```
logical DatabaseHandle :: Open (DictionaryHandle &dict_handle,
                char *cpath, PIACC accopt, logical w_netopt,
                logical online_version, uint16 version_nr, Re-
                sourceTypes local_ressources, char sysenv )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dict_handle	Dictionary handle The dictionary handle usually refers to an opened dictionary. To check whether a dictionary is opened you can use the !-operator.
cpath	Complete path The complete path is passed as 0-terminated string with a maximum length of 255 characters.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).
w_netopt	Multi-user option YES indicates that multi-user access is requested. NO indicates exclusive use of database. Accessing a database in update or write mode, NO guarantees absolute exclusive access.
online_version	Online versioning option When this option is set the database will be enabled vor online versioning. When the option is set to NO the system variable ONLINE_VERSION is checked instead. Default: NO
version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION
local_ressources	Resource type

Depending on the resource type the database or dictionary is opened on the client or server side.

RES_automatic

When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side.

RES_local

The dictionary or database will be opened on the client machine in any case.

RES_server

The dictionary or database will be opened on the server machine side in any case.

sysenv System application

This option indicates that the application is running as system application. In this case context functions are disabled and will not be executed. This option should never be set in normal applications because this may lead to logical inconsistence of the database.

d1 - Create new database

This function allows creating a new database. Usually creating a database explicitly is not necessary. When, however, special options as low est and higest local identifiers (LOID) are to be passed this constructor can be used.

```
logical DatabaseHandle :: Open (char *cpath, int16 lowEBN, int16
                   highEBN, int32 dasize, logical largedb, logi-
                   cal pindep )
```

Return value The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

cpath Complete path

The complete path is passed as 0-terminated string with a maximum length of 255 characters.

lowEBN	First entry number in database Low range number for the mainbase. Depending on the database (small or large) the range number is between 0 and 252 (small DB) or 0 and 32767 (0x7fff) for large databases.
highEBN	Last entry number in database High range number for the mainbase. Depending on the database type the range number is between 0 and 252 (small DB) or 0 and 32767 (0x7fff) for large databases.
dasize	Size for data area The data area size allows limiting the area for the data area. When no data area is passed (UNDEF), the data area expands whenever more space is needed.
largedb	Large database option The large database option idicates that a large database is to be defined. This information is stored in the database header after creating the database.
pindep	Platform independance option The plattform independance option idicates that integer numbers are to be stored in platform independent format. This information is stored in the database header after creating the database.

d2 - Creates database handle for dictionary

The function creates a database handle from the dictionary handle.

```
logical DatabaseHandle :: Open (DictionaryHandle &dict_handle,
                            PIACC accopt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dict_handle	Dictionary handle The dictionary handle usually refers to an opened dictionary. To check whether a dictionary is opened you can use the !-operator.
accopt	Access option

The access option defines the way instances in a property handle are to be accessed (read, update, write).

OpenRecovery - Open recovery file

The function opens the recovery file. Usually this is done automatically when opening the database and should not be opened explicitely by the user.

```
logical DatabaseHandle :: OpenRecovery (char *userinfo, int16
                 uilen )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
userinfo	Area for application recovery information This area can be provided by the application program and is expected to contain application data. When writing an entry to the recovery file the information is copied to the recovery entrie's user area. The application may change the conten of the area but not the location as long as the recovery file is opened.
uilen	Length of application data area The lenght describes the length of the application data area provided as **serinfo**.

OpenWorkspace - Open Workspace

The function creates or opens an existing workspace. After opening the workspace all updates are stored in the opened workspace. When the workspace is used the first time it is created automatically. When it does already exist the existing workspace is opened. You can check whether a workspace exists using the LocateWorkspace() function, which returns true when the workspace has already been created.

Usually the workspace file is created in the same folder as the database. You may, however, pass an explicit location for the workspace via the ws_path parameter.

```
logical DatabaseHandle :: OpenWorkspace (char *ws_names, char
                 *user_name, logical exclusive, char *ws_path )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
ws_names	Workspace name The workspace name is the extension of the current workspace or database. The database can be considered as the root for all workspaces. The workspace name may address a workspace on top of the current one (simple workspace name) or a workspace on any higher level by passing a sequence of workspace names separated by '.'.
user_name	User name When accessing user protected resources as databases or workspaces, a user must be passed as 0-terminated string, otherwise NULL.
exclusive	
ws_path	Physical location for the workspace The physical location must be accessible from the server, not from the client. When running in a client/server environment the client application should not path locations directly but rather via symboloc file names (file catalogue).

RecreateExtent - Recreate Index for an extent

The function repairs the indexes for a corrupted extent index. The function deletes all indexes for the extent and parses the database for instances with the type of the extent. The function works correct only, when all instances in the database belong to the extent.

```
logical DatabaseHandle :: RecreateExtent (char *extnames )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
extnames	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

operator bool - Database handle opened?

The function returns YES (true) when the database jandle is opened and NO (false) when the databse is not opened or when an error had occured while constructing the database handle.

```
NOTYPE DatabaseHandle :: operator bool ( ) const
```

Return value

operator!= - Compare database handles

The function returns true (YES) when the database handles refer to different database access blocks and false (NO) otherwise.

i00

```
logical DatabaseHandle :: operator!= (DatabaseHandle
                        &dbhandle_ref )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
dbhandle_ref	

i01

```
logical DatabaseHandle :: operator!= (DBObjectHandle
                        &obhandle_ref )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
obhandle_ref	

operator= - Assignment operator

The operator assigns the database access block of the passed database handle to the current database handle. Before the current database handle is closed.

i00

```
DatabaseHandle &DatabaseHandle :: operator= (const DatabaseHan-
                        dle &dbhandle_refc )
```

Return value	This is pointer to an opened database handle. The database handle can be an opened database handle DatabaseHandle as well as an opened dictionary handle (DictionaryHandle).
dbhandle	Pointer to database handle This is pointer to an opened database handle. The database handle can be an opened database handle DatabaseHandle as well as an opened dictionary handle (DictionaryHandle).

i01

```
DatabaseHandle &DatabaseHandle :: operator= (ACObject *acobject
                 )
```

Return value	This is pointer to an opened database handle. The database handle can be an opened database handle DatabaseHandle as well as an opened dictionary handle (DictionaryHandle).
acobject	

~DatabaseHandle - Destructur

The destructor closes the database handle. Closing the database handle will reduce the use count. The internal resources, the database access block is removed, when the use count becomes 0, i.e. when the last database handle referring to this resource is closed or destroyed.

```
                        DatabaseHandle :: ~DatabaseHandle ( )
```

DictionaryHandle - Dictionary Handle

The dictionary handle is used for providing schema definitions from the dictionary. The dictionary creates internal images from the externally stored schema definitions. These internal images (**{.r DBStructDef**}) can be provided by means of dictionary functions.

Because the dictionary is a database handle **{.r DBHandle**} you can access schema information also directly via PI functions.

BaseType - Returns internal number for elementary types

The function returns the internal number for elementary types (STRING, CHAR, INT,...). If the type name (strnames) passed is not a supported basic type the function returns UNDEF.

```
int16 DictionaryHandle :: BaseType (char *strnames )
```

Return value	For user-defined types (structures or enumerations) the internal type identification (number) is returned. If the type is unknown the function returns UNDEF (0). For elementary types (basic types -> {.r DataTypes}) the value is negative.
strnames	Structure name The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.

CheckExpression - Check expression syntax

The function terurns YES when the expression is invalid or no valid object handle has been passed.

```
logical DictionaryHandle :: CheckExpression (char *expression,
                     DBObjectHandle &dbobj_handle, char *clsnames )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
expression	OQL expression

An OQL expression defines a condition according to the OQL syntax. OQL expressions must always terminate with ';'. The OQL-Expression is passed as 0-terminated string.

dbobj_handle

clsnames

CopyCodeset - Copy enumeration

The function copies an enumeration (Codeset) from one dictionary to another.

```
logical DictionaryHandle :: CopyCodeset (DictionaryHandle
                  &srce_dicthandle, char *strname, char
                  *newnames, PIREPL db_replace, logical retain-
                  SID, logical retain_schemav )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
srce_dicthandle	
strname	Type name The type name is passed as 0-terminated string or as buffer with maximum 40 characters filled with trailing blanks.
newnames	New name for an extent or type The new name must be passed only if the type is to be renamed. The name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
db_replace	
retainSID	Retain internal type numbers If this option is set to YES the function trys to re-use the internal type number from the source dictionary. If this is not possible the type gets a new number in the target dictionary. This option is used normally only when copying a complete dictionary.
retain_schemav	

CopyExtentDef - Copy extent definition

The function copies an extent definition from one dictionary to another.

```
logical DictionaryHandle :: CopyExtentDef (DictionaryHandle
                    &srce_dicthandle, char *extentname, char
                    *newnames, char *targ_struct, logical transac-
                    tion, logical retain_schemav )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
srce_dicthandle	
extentname	Extent name The name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
newnames	New name for an extent or type The new name must be passed only if the type is to be renamed. The name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
targ_struct	Target type The target type must be passed when the type name for the extent has been changed (e.g. because of a copy/rename operation). The name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
transaction	Transaction option When passing YES the function creates a transaction while copying the extent definition.
retain_schemav	

CopyStructure - Copy structure definition

The function copies a structure definition from one dictionary to another.

```
logical DictionaryHandle :: CopyStructure (DictionaryHandle
                    &srce_dicthandle, char *strname, char
                    *newnames, char *topname, PIREPL db_replace,
                    logical retainSID, logical retain_schemav )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
srce_dicthandle	
strname	Type name The type name is passed as 0-terminated string or as buffer with maximum 40 characters filled with trailing blanks.
newnames	New name for an extent or type The new name must be passed only if the type is to be renamed. The name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
topname	Name of top-type When calling the function this field should contain the name of the type to be copied. When processing recursive copy operations the name is used to avoid recursion while copying.
db_replace	
retainSID	Retain internal type numbers If this option is set to YES the function trys to re-use the internal type number from the source dictionary. If this is not possible the type gets a new number in the target dictionary. This option is used normally only when copying a complete dictionary.
retain_schemav	

CopyType - Copy type definition

The function copies a type definition from one dictionary to another.

```
logical DictionaryHandle :: CopyType (DictionaryHandle
                 &srce_dicthandle, char *strnames, char
                 *newnames, char *topname, PIREPL db_replace,
                 logical retainSID, logical transaction, logi-
                 cal retain_schemav )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
srce_dicthandle	
strnames	Structure name The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.
newnames	New name for an extent or type The new name must be passed only if the type is to be renamed. The name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
topname	Name of top-type When calling the function this field should contain the name of the type to be copied. When processing recursive copy operations the name is used to avoid recursion while copying.
db_replace	
retainSID	Retain internal type numbers If this option is set to YES the function trys to re-use the internal type number from the source dictionary. If this is not possible the type gets a new number in the target dictionary. This option is used normally only when copying a complete dictionary.
transaction	Transaction option When passing YES the function creates a transaction while copying the extent definition.
retain_schemav	

CreateEnum - Create new enumeration

he function creates a new enumeration. The dictionary must be opened in write mode.

When defining a new enumeration in a dictionary it has to be created before it can be opened for adding the enumeration items.

```
logical DictionaryHandle :: CreateEnum (char *enum_name, char
                    *basetype )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
enum_name	Enumeration name The enumeration name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
basetype	

CreateTempExtent - Create temporary extent

A temporary extent can be created for storing results of a qeuery (e.g. a selection) within an application. Temporary extents are created in main storage or in a temporary database and are available as long as the database handle is opened. They will be removed automatically when closing the database handle.

When a temporary extent has been created once, you can open any number of property handles for accessing the extent.

You can define an extent for a structure definition defined in the external dictionary by referring to the tsructure name) or by an internal structure definition that has been created by the application and is referenced by the field definition passed to the function.

ci

```
char *DictionaryHandle :: CreateTempExtent (char *strnames, char
                    *extnames_w, char *key_name_w, char
                    *baseexts_w, logical weak_opt_w, logical
                    own_opt_w )
```

Return value	The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
strnames	Structure name

	The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.
extnames_w	Extent name
	The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
key_name_w	Key name for conversion
	The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks. If no key name is passed (NULL) the sort key according to the selected sort order is used instead,
baseexts_w	Name for base extent
	A base extent or base collection can be passed that defines a superset for the temporary extent. The extent name is passed as 0-terminated string with maximum 40 characters.
weak_opt_w	Weak-typed option
	This option must be true (YES) when a collection may refer to instances of differet types, wich are based on the same base structure.
own_opt_w	Owning collection
	This option must be set to true (YES) if the collection owns the instances it is referring to. In this case the collection may not refer to instances from other collections. Removing instances from an owning collection will result in deleting the instance completely.

i01

```
char *DictionaryHandle :: CreateTempExtent (DBFieldDef
                  *field_def, char *extnames_w )
```

Return value	The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
field_def	Property definition
	The property defintion contains the metadata for the referenced property instance..

extnames_w	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

DeleteEnum - Delete enumeration definition

This function deletes an enumeration definition from the external dictionary.

```
logical DictionaryHandle :: DeleteEnum (char *enum_name )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
enum_name	Enumeration name The enumeration name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

DictionaryHandle - Create dictionary handle

Usually the dictionary is created as local dictionary, i.e. the external dictionary must be provided in the local environment.

When the application wants to refer to databases on a server the dictionary has to be opened as server dictionary as well using the external dictionary on the server. In this case an ODABAClient with an opened connection has to be passed. When the connection is not opened the system tries to open the dictionary as local dictionary.

ci0

```
     DictionaryHandle :: DictionaryHandle
(ODABAClient &odaba_client, char *cpath, PIACC
accept, logical w_netopt, uint16 version_nr,
ResourceTypes local_ressources, char sysenv )
```

odaba_client	ODABA Client Handle The ODABA client handle can be passes as connectet or ea empty handle.

cpath	Complete path The complete path is passed as 0-terminated string with a maximum length of 255 characters.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).
w_netopt	Multi-user option YES indicates that multi-user access is requested. NO indicates exclusive use of database. Accessing a database in update or write mode, NO guarantees absolute exclusive access.
version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION
local_ressources	Resource type Depending on the resource type the database or dictionary is opened on the client or server side. **RES_automatic** When a connection is opened to the server the dictionary is opened on the server side when passing a symbolic database path (like %DB_PATH%). When passing a dictionary path the dictionary is opened on the client side. When no connection is opened the dictionary or database will be opened on the client side. **RES_local** The dictionary or database will be opened on the client machine in any case. **RES_server** The dictionary or database will be opened on the server machine side in any case.
sysenv	System application

This option indicates that the application is running as system application. In this case context functions are disabled and will not be executed. This option should never be set in normal applications because this may lead to logical inconsistence of the database.

i02

```
                         DictionaryHandle :: DictionaryHandle (
                    )
```

i03

```
                         DictionaryHandle :: DictionaryHandle
                    (Dictionary *_dictionary )
```

_dictionary

i04

```
                         DictionaryHandle :: DictionaryHandle
                    (const DictionaryHandle &dictionary_refc )
```

dicthdl

i1

```
                         DictionaryHandle :: DictionaryHandle
                    (DatabaseHandle &db_handle )
```

db_handle	Pointer to database handle This is pointer to an opened database handle. The database handle can be an opened database handle DatabaseHandle as well as an opened dictionary handle (DictionaryHandle).

GetExtentDef - Get extent definition

The function returns the extend definition for the passed extent name from the internal dictionary. When the extent definition has not been found in the internal dictionary the function will not read the extent definition from the external dictionary (see ProvideExtendDef()).

```
DBExtend *DictionaryHandle :: GetExtentDef (char *extname )
```

Return value	
extname	Extent name

The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

GetID_SIZE - Size for identifying names in ODABA

ODABA has a unique size for identifying names. Since the identifier size may change between different ODABA versions this function returns the identifier size for the current version.

```
int16 DictionaryHandle :: GetID_SIZE ( )
```

Return value	Size of the instance or property area.

GetTempName - Get unique name for temporary resource

The function provides a unique internal name that can be used for creating temporary extents or other resources.

```
char *DictionaryHandle :: GetTempName (char *extnames_w )
```

Return value	The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
extnames_w	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

IsBasicType - Is type an elementary type?

The function returns YES when the passed type is one of the elementary ODABA data types.

```
logical DictionaryHandle :: IsBasicType (char *typenames )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
typenames	Type name The type name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

ProvideExtentDef - Provide extent definition

The function returns the extend definition for the passed extent name from the internal dictionary. When the extent definition has not been found in the internal dictionary the function will provide the extent definition in the internal dictionary by reading it from the external dictionary.

```
DBExtend *DictionaryHandle :: ProvideExtentDef (char *extnames )
```

Return value	
extnames	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

ProvideStructureDef - Provide structure definition from internal or external dictionary

The function returns the structure definition for the passed structure name from the internal dictionary. When the structure definition has not been found in the internal dictionary the function will provide the extent definition in the internal dictionary by reading it from the external dictionary.

```
DBStructDef *DictionaryHandle :: ProvideStructureDef (char
                *strnames )
```

Return value	The structure definition is provided in the internal format as pointer to a DBStructDef object.
strnames	Structure name The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.

operator bool - Dictionary opened

The function returns YES (true) when the dictionary handle is opened and NO (false) when the dictionary is not opened or when an error had occured while constructing the dictionary handle.

```
NOTYPE DictionaryHandle :: operator bool ( ) const
```

Return value	

operator= -

```
DictionaryHandle &DictionaryHandle :: operator= (const Diction-
                    aryHandle &dictionary_refc )
```

Return value	
dicthandle	

operator== - Compare dictionary handles

The function returns YES (true) when the dictionary handles compared are identical.

```
logical DictionaryHandle :: operator== (const DictionaryHandle
                    &dictionary_refc )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
dict_handle	Dictionary handle The dictionary handle usually refers to an opened dictionary. To check whether a dictionary is opened you can use the !-operator.

~DictionaryHandle - Destructor

```
                            DictionaryHandle :: ~DictionaryHandle (
                    )
```

Error - General Error object

The error object is used to store and pass error information to the application. Errors are identified by error class and eror number. In addition the class and function name detecting the problem and a short error explanation can be provided. Moreover, an error may include upto 6 context depending error variables that can be displayed in the error message.

Usually error messages are written to a log file (error.lst) which is stored in a folder addressed by the TRACE environment or ini-file variable. It is, however, also possible to display errors on the terminal.

Usually errors should be reset in all functions that may signal an error. Otherwise the calling function may not be able to determine whether the error signaled is an old error or has just been signaled in the called function. This strategy requires, on the other hand, that signaled errors have to be saved when other functions are called in the error handling thet might generate errors again, since those functions will reset the error. You can use the Copy() function to save the error.

The way errors are presented in the application depends on the error handler installed (ErroerHandle). Usually errors are written to the console output for console applications and shown in a message box for windows applications.

CheckError - Check error state

The function checks whether an error is set in the error object and returns the error number, if so.

```
int32 Error :: CheckError ( )
```

Return value

Copy - Copy error

The function copies the error object to save relevant error information. You can use the function to save error information that might be destroyed when calling other functions.

```
void Error :: Copy (Error &err_obj )
```

err_obj	Error object The error object contains information about the last error detected.

CreateExceptions - Throw exception

The function enables exception throwing, i.e. an exception is thrown, when an error is signaled. Usully, no exception is thrown.

```
void Error :: CreateExceptions (logical exceptions )
```

exceptions	Trow exception When the option is set to YES (true) exceptions are trown.

DisplayMessage - Dispaly message

The function allows displaying a message for a signaled or passed error code. Depending on the error heandle set for the error the error is written to the console or displayed on the terminal.

i0 - Display user error

The function displays an error passed by the user. The passed error variables are inserted for the place holders in the error message definition. When the error code passed is 0 the last signaled error code is used instead.

```
logical Error :: DisplayMessage (const int16 err_code, char
                 *errvar1, char *errvar2, char *errvar3, char
                 *errvar4, char *errvar5, char *errvar6 )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
err_code	Error code The error code passed must be a defined error code.
errvar1	First error variable

	The text for the first error variable will replace the place holder %1 or the first occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar2	Second error variable The text for the second error variable will replace the place holder %2 or the seond occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar3	Third error variable The text for the third error variable will replace the place holder %3 or the third occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar4	Fourth error variable The text for the fourth error variable will replace the place holder %4 or the fourth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar5	Fifth error variable The text for the fifth error variable will replace the place holder %5 or the fifth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar6	Sixth error variable The text for the sixth error variable will replace the place holder %6 or the sixth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.

i1 - Display current error

Usually error messages are written to the error log, only. This function shows the error in the given application context (console or message box).

```
logical Error :: DisplayMessage ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

i2 - Display application error with defined error database

The function displays an error passed by the user. The passed error variables are inserted for the place holders in the error message definition. When the error code passed is 0 the last signaled error code is used instead. The function retrieves the error message from the error source (usually an error database) passed to the error calling the ErrorHandle::GetError() function.

```
logical Error :: DisplayMessage (void *error_source, const int16
                  err_code, char *errvar1, char *errvar2, char
                  *errvar3, char *errvar4, char *errvar5, char
                  *errvar6 )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
error_source	
err_code	Error code The error code passed must be a defined error code.
errvar1	First error variable The text for the first error variable will replace the place holder %1 or the first occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar2	Second error variable The text for the second error variable will replace the place holder %2 or the seond occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar3	Third error variable The text for the third error variable will replace the place holder %3 or the third occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.

errvar4	Fourth error variable The text for the fourth error variable will replace the place holder %4 or the fourth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar5	Fifth error variable The text for the fifth error variable will replace the place holder %5 or the fifth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar6	Sixth error variable The text for the sixth error variable will replace the place holder %6 or the sixth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.

i3 -

The function retrieves the error message from the error source (usually an error database) passed to the error calling the ErrorHandle::GetError() function and displays the message in the application context. The way the error message is displayed depends on the ErrorHandle::DisplayMessage() function of the error handle associated with the error.

```
logical Error :: DisplayMessage (void *error_source )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
error_source	

Error - Constructor

The function constructs an error object. The function does not set an error handle. This must be done explicitly by the application, which can set an application specific error handle.

i0 - Application error

The constructor creates an application or subsystem error that can be used to handle signaled errors in the application.

```
     Error :: Error (int16 metacode, char
*pid, char *errclass, char *ttext )
```

metacode	Meta code for the error When using errors in an hirarchical application the meta error indicates which system has caused an error. Thus, e.g. a database error is passed to the application error with a default error code for indicating a "database error". In this case more detailed information can be retrieved from the last set database error, which is displayed than instead of the more general database error. Usually application errors will not define a meta-code. Only when creating a subsystem with a separate error object a meta code has to be assigned for errors in this sub-system and handles by the error handle of the sub system.
pid	
errclass	Error class Errors are grouped in error classes. An error class is defined for each subsystem or application. In database applications the error class defines the extent that contains the error descriptions for all errors of the application or subsystem.

i1 - Dummi constructor

The constructor creates an empty error object.

```
     Error :: Error ( )
```

i2 - Copy Constructor

The function copies the information of the passed error object into the newly created error object and can be used instaed of the Copy() function.

```
     Error :: Error (Error &err_obj )
```

err_obj	Error object The error object contains information about the last error detected.

GetDecision - Ask for user decision

The function creates an message from the error code and the passed error variables and generates a decision that is displayed in the specific application context (console message for console applications and decision box for windows applications. To execute the function successfully an error handle should be set. If not, a simple error handle will be constructed.

i0 - Display application decision

The function displays a decision forced by the application. The passed error variables are inserted for the place holders in the error message definition. The decision text is taken from the eror description for the passed error code as defined in the error source set for the active arror handle.

```
logical Error :: GetDecision (const int16 err_code, char
                 *errvar1, char *errvar2, char *errvar3, char
                 *errvar4, char *errvar5, char *errvar6 )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
err_code	Error code The error code passed must be a defined error code.
errvar1	First error variable The text for the first error variable will replace the place holder %1 or the first occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar2	Second error variable The text for the second error variable will replace the place holder %2 or the seond occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar3	Third error variable

	The text for the third error variable will replace the place holder %3 or the third occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar4	Fourth error variable The text for the fourth error variable will replace the place holder %4 or the fourth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar5	Fifth error variable The text for the fifth error variable will replace the place holder %5 or the fifth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar6	Sixth error variable The text for the sixth error variable will replace the place holder %6 or the sixth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.

i1 - Display predefined decision

The function displays a decision with the error text set in the text field of the error object. This function shows the decision in the given application context (console or message box).

```
logical Error :: GetDecision ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

i2 - Display application decision with different error source

The function displays a decision forced by the application. The decision text is constructed from the error definition read from the passed error source replacing the place holders by the passed error variables.

```
logical Error :: GetDecision (void *error_source, const int16
                  err_code, char *errvar1, char *errvar2, char
                  *errvar3, char *errvar4, char *errvar5, char
                  *errvar6 )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
error_source	
err_code	Error code The error code passed must be a defined error code.
errvar1	First error variable The text for the first error variable will replace the place holder %1 or the first occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar2	Second error variable The text for the second error variable will replace the place holder %2 or the seond occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar3	Third error variable The text for the third error variable will replace the place holder %3 or the third occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar4	Fourth error variable The text for the fourth error variable will replace the place holder %4 or the fourth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar5	Fifth error variable The text for the fifth error variable will replace the place holder %5 or the fifth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar6	Sixth error variable

The text for the sixth error variable will replace the place holder %6 or the sixth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.

i3 - Display decision with different error source

The function displays a decision forced by the application. The decision text is constructed from the error definition read from the passed error source replacing the place holders by the error variables set in the error (err_var1... 6).

```
logical Error :: GetDecision (void *error_source )
```

Return value

The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

error_source

GetErrorHelpID - Get help context id

The function returns a help context id that can be used to call an online help topic associated with the error.

```
int32 Error :: GetErrorHelpID ( )
```

Return value

GetErrorText - Get Error text

The function creates an error message from the error definition read from the passed error source replacing the place holders by the passed error variables.

```
char *Error :: GetErrorText (void *error_source, const int16
                 err_code, char *errvar1, char *errvar2, char
                 *errvar3, char *errvar4, char *errvar5, char
                 *errvar6 )
```

Return value

The error text is passed as 0-terminated string with a maximum length of 500 characters.

error_source

err_code

Error code

The error code passed must be a defined error code.

errvar1	First error variable The text for the first error variable will replace the place holder %1 or the first occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar2	Second error variable The text for the second error variable will replace the place holder %2 or the seond occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar3	Third error variable The text for the third error variable will replace the place holder %3 or the third occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar4	Fourth error variable The text for the fourth error variable will replace the place holder %4 or the fourth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar5	Fifth error variable The text for the fifth error variable will replace the place holder %5 or the fifth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar6	Sixth error variable The text for the sixth error variable will replace the place holder %6 or the sixth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.

GetText - Get error text

The function returns the error text currently set in the error objkect.

```
char *Error :: GetText ( )
```

GetTitle - Get title

The function returns the eror title for the error class.

```
char *Error :: GetTitle ( )
```

Initialize - Initialize error

The function initializes an error object.

i0 - Reset error object

The function will reset all attributes in the error objects to its initial values.

```
void Error :: Initialize ( )
```

i01 - Initialize error object

The function wil reset the orror specification with the parameters passed.

```
void Error :: Initialize (int16 metacode, char *pid, char
                          *errclass, char *ttext )
```

metacode — Meta code for the error

When using errors in an hirarchical application the meta error indicates which system has caused an error. Thus, e.g. a database error is passed to the application error with a default error code for indicating a "database error". In this case more detailed information can be retrieved from the last set database error, which is displayed than instead of the more general database error. Usually application errors will not define a meta-code. Only when creating a subsystem with a separate error object a meta code has to be assigned for errors in this sub-system and handles by the error handle of the sub system.

pid

errclass — Error class

Errors are grouped in error classes. An error class is defined for each subsystem or application. In database applications the error class defines the extent that contains the error descriptions for all errors of the application or subsystem.

InsertStatField - Insert status line field

The function creates a field in the status line for displaying information from the error object in the status line. The behaviour of the statusline depends on the handling in the error handle. The default error handle does not support staus line information.

```
void Error :: InsertStatField ( )
```

RemoveStatField - Remove user field from the status line

The function removes a statusline field that had been inserted before using InserStatField().

```
void Error :: RemoveStatField ( )
```

Reset - Reset error text

The function resets the error text and the error type, but not the error code. For resetting the error object call ResetError().

```
void Error :: Reset ( )
```

ResetAllErrors - Reset all errors

The function resets the erors for all subsystems for the given thread.

```
void Error :: ResetAllErrors ( )
```

ResetError - Reset error object

The function resets the current error settings. This function should be called in any function that might set an error.

```
void Error :: ResetError ( )
```

SetError - Signal error

The function signals an error for the error object. Usually the error is recorded in a log file (error.lst).

```
void Error :: SetError (const int16 err_code, char *obj, char
                        *mod )
```

err_code	Error code The error code passed must be a defined error code.
obj	Object or class name The class name of the function that has detected the error is passed as 0-terminated string.
mod	Module or function The module or function name where the error was detected is passed as 0-terminated string.

SetErrorVariable - Set error variable

The function is used to set an error variable before signaling an error (-> SetError()). The value of this error variable will replace the place holder according to the variable number in the error description text (e.g. setting error variable 2 will replace the place holder %2 or the second occurence of %s in the error text).

```
void Error :: SetErrorVariable (int8 varnum, char *vartext,
                                int16 varlen )
```

varnum	Variable number Number of the error variable to be set.
vartext	Variable text The text for the variable is passed as 0-terminated string with a maximum length of 80.
varlen	Variable length When the variable text is not passed as 0-terminated string the length defines the length for the string passed.

SetHandle - Set error handle

The function allows setting an application specific error handle for the error object.

```
void Error :: SetHandle (ErrorHandle *error_hdl )
```

error_hdl	Error handle An error handle is usually passed as application specific error handle that provides application or subsystem specific functions for displaying errors.

SetLanguage - Select language for error messages

The function allows setting a language for displaying errors when the associated error handle supports multilingual error messages. The exact language definitions are specific for the associated handler, however, the English language name is used in most cases.

```
void Error :: SetLanguage (char *err_lang )
```

err_lang	Error language

The language is passed as 0-terminated string. The exact language definitions are specific for the associated handler, however, the English language name is used in most cases.

SetSource - Set error resource

When supporting an error resource that contains the error definitions this can be associated with the error using this function. The type of the error resource depends on the eror handle associated with the error. Usually, database applications pass a database handle for a database that contains an extent with the error class name that stores the error definitions.

```
void Error :: SetSource (void *error_source )
```

error_source

SetStatField - Set value in status line

When a status field has been inserted in the status line of the application (-> InsertStatField()), the function will send the passed string value to the status line by means of the error handle.

```
void Error :: SetStatField (char *string )
```

string	String area
	Pointer to the 0-terminated string area.

SetStatText - Set status line text

The function will send the passed string value to the default text field of the status line by means of the error handle. This function does not require a application defined field in the status line as provided with InsertStatField().

```
void Error :: SetStatText (char *string )
```

string	String area
	Pointer to the 0-terminated string area.

SetText - Set text

The function sets the text with an error message in the error object text field.

```
void Error :: SetText (char *err_text )
```

err_text	Error text The error text is passed as 0-terminated string with a maximum length of 500 characters.

SetTitle - Set error object title

The function changes the title for the error object class.

```
void Error :: SetTitle (char *ttext )
```

SetTracePath - Set path for error-log file

The function changes the path for the current error log-file. The default ErrorHandle records all errors in a file error.lst which is located in a folder addressed by the path defined in the environment or system variable TRACE. The function will change the settings of this system variable.

```
void Error :: SetTracePath (char *cpath )
```

cpath	Complete path The complete path is passed as 0-terminated string with a maximum length of 255 characters.

SetType - Set eror type

The function sets the passed error type. The error type is evaluated by the associated error handle for displaying error messages in an appropriate way.

```
void Error :: SetType (char err_type )
```

err_type

SetupErrText - Setup error text

The function replaces the place holders in the eror text with the error variables set in the error object.

```
void Error :: SetupErrText (void *error_source )
```

error_source

TraceMessage - Write message to log-file

The function allows writing a message to the log file without signaling an error. The error variables passed are replaced in the message befor writing the message to the log-file.

```
void Error :: TraceMessage (char *errvar1, char *errvar2, char
                  *errvar3, char *errvar4, char *errvar5, char
                  *errvar6 )
```

errvar1	First error variable The text for the first error variable will replace the place holder %1 or the first occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar2	Second error variable The text for the second error variable will replace the place holder %2 or the seond occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar3	Third error variable The text for the third error variable will replace the place holder %3 or the third occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar4	Fourth error variable The text for the fourth error variable will replace the place holder %4 or the fourth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar5	Fifth error variable The text for the fifth error variable will replace the place holder %5 or the fifth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.
errvar6	Sixth error variable The text for the sixth error variable will replace the place holder %6 or the sixth occurence of %s in the error message. The error variable is passed as 0-terminated string with a maximum length of 80.

operator= - Assign error object

The function assigns all attributes of the error object passed to the current error object. It can be used instaed of the Copy() function.

```
Error &Error :: operator= (Error &err_obj )
```

Return value	The error object contains information about the last error detected.
err_obj	Error object The error object contains information about the last error detected.

~Error - Destructor

The function destroys the error object.

```
Error :: ~Error ( )
```

EventHandler - Event Handler Class

The Event Handler Class is a base class for supporting writing event handlers. It provides some basic functionality for setting and calling event handlers for handling server events.

You may derive your own handler classes from EventHandler to provide handler functions for server events. You may overload the handler functions InstanceEventHandler() and PropertyEventHandler() for providing your application specific event handling.

The event handler allows handling instance, property (collection) or local events. Instance and property events are client server events that are generated, when an instance or collection changes. Local events are those events, which are usually handled in the instance or property context. You may, however, set event handler for local events for a specific property handle, which allows overwriting or expanding context functions.

ActivateProcessEventHandler - Activate process event handlers

The function activates the event handlers for process events. When not activating process event handling process events will not passed to the application.

```
void EventHandler :: ActivateProcessEventHandler (
                                  )
```

ActivateServerEventHandler - Activate server event handlers

The function activates the event handlers for server events. When not activating server event handling process events will not passed to the application.

```
void EventHandler :: ActivateServerEventHandler (
                                  )
```

EventHandler - Konstruktor

Constructing an event handler class instance for the property handle passed to the function. The constructor sets the property event handler as well as the instance event handler. The property handle is registered for receiving server events (-> RegisterHandle()).

```
                                         EventHandler :: EventHandler (Proper-
                                  tyHandle &prop_hdl )
```

prop_hdl	Property Handle

Is a reference to an (usually) opened property handle.

InstanceEventHandler - Instance event handler

The instance event handler has to be overloaded when specific handling for instance events as updated or deleted has to be provided. The type of event is passed via the event_id. The objid refers to the instance identity of the updated instance.

When a notification handler is implemented in the context class, it will be called after calling the event handler set for the ptoperty handle.

```
logical EventHandler :: InstanceEventHandler (CSA_Events
                        event_id, int32 objid )
```

Return value	When this value is true the function will continue, otherwise the processing terminates.
event_id	Ivend type The event type defines the type of the passed event.
objid	Local object identity (LOID) The local object identity is a 31-bit number that identifies an object instance uniquely in a database. LOIDs are available only for independently stored instances but not for instances of imbedded structures.

ProcessInstanceHandler - Process Instance Event Handler

The process event handler has to be overloaded when specific handling for process events (as update or delete instance) has to be provided. The type of event is passed via the intevent parameter.

The function should return true (YES) to pre-process handlers to abort the process.

```
logical EventHandler :: ProcessInstanceHandler (DB_Event inte-
                        vent )
```

Return value	When this value is true the function will continue, otherwise the processing terminates.
intevent	Event identifier The event identifier is an internal number that is defined for typical events.

ProcessPropertyHandler - Process Property Event Handler

The process event handler has to be overloaded when specific handling for process events (as read or change selection) has to be provided. The type of event is passed via the intevent parameter.

The function should return true (YES) to pre-process handlers to abort the process.

```
logical EventHandler :: ProcessPropertyHandler (DB_Event inte-
                         vent )
```

Return value	When this value is true the function will continue, otherwise the processing terminates.
intevent	Event identifier The event identifier is an internal number that is defined for typical events.

PropertyEventHandler - Property event handler

The property event handler has to be overloaded when specific handling for property (collection) events as updated or deleted has to be provided. The type of event is passed via the event_id. The objid refers to the index identity of the updated collection.

When a notification handler is implemented in the context class, it will be called after calling the event handler set for the ptoperty handle.

```
logical EventHandler :: PropertyEventHandler (CSA_Events
                         event_id, int32 objid )
```

Return value	When this value is true the function will continue, otherwise the processing terminates.
event_id	Ivend type The event type defines the type of the passed event.
objid	Local object identity (LOID) The local object identity is a 31-bit number that identifies an object instance uniquely in a database. LOIDs are available only for independently stored instances but not for instances of imbedded structures.

~EventHandler - Destructor

Destructing the class will reset the handler in the property handle. The property handle is unregistered from receiving server events (-> UnregisterHandle()).

```
EventHandler :: ~EventHandler ( )
```

EventLink - Event Link

This is a function link object for handling events. The function link stores a pointer to the handler class instance and the function to be called.

The following status indicators are used:

stsini - handler is active and will be executed

EventLink - Constructor

The constructor creates an event link that defines a link to an event handler.

i0 - Event link

This constructor creates an event link. Instead of using the constructor directly the macro

ELINK(instptr, clsname, funcname)

should be used. This allows easily defining an event link passing the instance pointer, the class name and the handler function name.

The calling conventions for thr linked function are as

logical vcls::handler(CSA_Events event, long loid, PropertyHandle &ph)

```
EventLink :: EventLink (vcls *vclsptr, EVTP evtptri )
```

vclsptr	Virtual class pointer The virtual class pointer refers to any type of class derived from the virtual class.
evtptri	Event handler pointer The event handler pointer is a function pointer as: logical vcls::EVTP(CSA_Events, long, PropertyHandle &).

i01 - Dummy constructor

```
EventLink :: EventLink ( )
```
i02

```
                        EventLink :: EventLink (vcls *vclsptr,
                  EVTPL evtptril )
```

vclsptr	Virtual class pointer The virtual class pointer refers to any type of class derived from the virtual class.
evtptril	Event handler pointer for local events The event handler pointer is a function pointer as: logical vcls::EVTPL(DB_Events, PropertyHandle &).

IsActive -

```
logical EventLink :: IsActive ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

~EventLink - Destruktor

```
                        EventLink :: ~EventLink ( )
```

Instance - Instance Handle

Instance handles are used to pass and return structured database instances. Instead of an instance handle a (void *) area can be passed, that is automatically converted into an instance handle. The instance area is allocated and freed by the application.

Key - Key Handle

Key handles are used to pass and return keys. Instead of a key handle a (char *) area can be passed, that is automatically converted into a key. The key area is allocated and freed by the application.

GetData - Provide key area

The function returns the key instance area as (char *) pointer.

```
char *Key :: GetData ( )
```

Return value	The key area is structured according to the key definition (key smcb).

Key - Konstruktor

A key handle is contructed with the key area passed to the handle.

i0

```
Key :: Key (char *keyarea )
```

keyarea	Key area The key area is structured according to the key definition (key smcb).

i01

```
Key :: Key ( )
```

SetData - Set key area

The function allows assigning a new key area to the key handle.

```
char *Key :: SetData (char *keyarea )
```

Return value	The key area is structured according to the key definition (key smcb).
keyarea	Key area The key area is structured according to the key definition (key smcb).

operator char* - Type conversion

The operator supports implicite type conversion from (char *) pointers into key handles.

```
NOTYPE Key :: operator char* ( )
```

Return value	

operator& - Adress operator

The operator returns the key area.

```
char *Key :: operator& ( )
```

Return value	The key area is structured according to the key definition (key smcb).

operator= - Assignment operator

The operator allows assigning a new key area from the passed key handle to the key handle.

```
Key &Key :: operator= (const Key &key_refc )
```

Return value	Reference to a key handle.
key_ref	Kea reference Reference to a key handle.

ODABAClient - ODABA client

To run client server applications you must create a ODABA client instance. To support several connections to different servers you can create one or more clients within your application.

When connecting to different servers you must create one client for each server. You can open several clients in an application. The first client, however, is considered to be the main client. The main client should be the last client closed in an application. After closing the main client you can open another main client. Since there is no hierarchy defined between clients the system will not check

The main client registers the process and activates the error log file. It opens the system database for providing error messages and the data catalogue if one has been specified in the system environment (see ODABAClient constructor). These information are described in an ini-file, which can be passed to the client.

For initializing and registring the process properly a client should be created also for locally running applications.

ActivateGUIMessages - Activate GUI-Messages

For console applications messages will be sent to the console, only. When messages should be displayed in GUI message boxes as well you can use the Activate GUIMessages() function to enable this feature.

```
void ODABAClient :: ActivateGUIMessages ( )
```

Connect - Connect to server

This function establishs a connection to the server. When not connecting the cleint to a server the client runs in local mode. When running in local mode all resources are located on th client machine.

When being connected to a server you can access ressources located on the server or on the local machine by setting the "local_resources"-parameter when constructing dictionary or database handles.

When connecting several times for closing the connection you must disconnect as often as you have connected to the server. When the client is connected once it cannot be connected to another server until the open connection is closed.

```
logical ODABAClient :: Connect (char *server_name, uint32
                 host_port )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
server_name	Server name The server name consists of the port number and the server identification. Both has been defined when starting up the server (e.g.6123@MetaServer). If no server string is passed the client expects the server name in an environment variable ODABA_SERVER or in an odaba.ini-File on the ODABA installation folder.
host_port	Port number The port number must be the same the server has been started with (e.g.6123). If no port number is passed the client expects the port number being defined in a system variable ODABA_SERVER_PORT or in a system environment INI-file on the ODABA installation folder.

Disconnect - Disconnect from server

Please make shure that all resources are closed before disconnecting the client. Disconnecting the client before closing all opened handles may cause problems and not all changes are stored.

```
void ODABAClient :: Disconnect ( )
```

Exist - Check whether a database exists

The function returns YES when a database with the given path exists and has been initialized as ODABA database. The database path may refer to a database or dictionary. When the referenced file does not exist or does not refer to an ODABA database the function returns NO.

Da die Datenbank zum Prüfen eröffnet wird, kann es im Fehlerfall geschehen, daß eine ungültige Datei angelegt wird.

```
logical ODABAClient :: Exist (char *cpath )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
cpath	Complete path The complete path is passed as 0-terminated string with a maximum length of 255 characters.

GetDBError - Get last database error

The function returns the last database error including error number and description. For more details see Error class definition.

```
Error *ODABAClient :: GetDBError ( )
```

Return value

GetDataSource - Get data source name

The function returns the data source name on position indx0. The function returns a value, only, if the application is working with a data catalogue, i.e. the ini-file must contain a valid DATA-CATALOGUE section.

```
char *ODABAClient :: GetDataSource (int32 indx0 )
```

Return value	The data source name is passed as 0-terminated string with a maximum length of 40 characters.
indx0	Position in collection The position in the collection addresses the first instance in the collection with 0. AUTO (-1) refers to an undefined position.

GetServerVariable - Get system variable from server

The function returns the value for a system variable set on the server side.

```
char *ODABAClient :: GetServerVariable (char *var_name )
```

Return value	The value for a system variable must not exceed 255 characters and is provided as 0-terminated string.
var_name	System variable name Name of the system variable on the server or client side. System variable names must not exceed 40 characters and are provided as 0-terminated strings.

IsConnected - Is client connected

The function checks whether the client is connected or not. When being connected the function returns YES, NO otherwise.

```
logical ODABAClient :: IsConnected ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

KillClient - Kill client on the server

The function allows killing one or all clients on the server. When killing all clients the client sending the command is not killed.

Befor killing the client(s) the system is waiting wait_sec seconds.

```
logical ODABAClient :: KillClient (int32 client_id, int32
                 wait_sec, logical send_message )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
client_id	Client the message is send to The client number (client_id) is a number the server has assigned to the client. You can retrieve client numbers by using the GetClientHandle() function which returns client information for all active clients.
wait_sec	Number of seconds to wait

The system waits the given number of seconds befor executing the request. Default is 300 seconds (5 minutes).

ODABAClient - Konstructor

To run an application in client/server mode at least one client must be constructed for an application. The first client created is the main client that should be created at the very beginning of the application and that should be closed at the very end of the application.

In addition a number of other clients can be created to connect to different servers in one application. When creating a main client an ini file can be provided to the constructor. This ini file defines the application section.

System and catalogue sections are read from the ODABA2.INI file that is stored in the ODABA2 installation path. It is, however, possible to provide separate system and catalogue definitions for the client with the passed INI-file. In this case the passed INI-file must contain either the system and catalogue sections or it must refer to an INI-file that contains these sections by defining the path for the system INI-file in the variable SYSTEM_ENVIRONMENT.

The INI-file passed to the client must contain a section with the name of the application or (when no application name has been passed) a section with the name "APPLICATION_DATA".

i00

```
      ODABAClient :: ODABAClient (char
*inipath, char *application_name, char
*progpath, ApplicationTypes application_type )
```

inipath	
application_name	Allication name The name of the application is usually also the section name for the application variables in the ini file.
progpath	Programme path

This is the path that is usually passed as first argument to the application.

Default: NULL

application_type	Run as console application This option indicates that the application will run as console application. In this case errors are sent to the console (default: YES). If this option is set to NO message boxes are created instead (for Windows, only).

i01

```
                ODABAClient :: ODABAClient ( )i02

                ODABAClient :: ODABAClient (const ODA-
            BAClient &client_refc )
```

client_ref

i03

```
                ODABAClient :: ODABAClient (CClient
            *cclient_ptr )
```

cclient_ptr

PackDatabase - Pack database

The function packs a database. The function packs the database by copying it to a new file. If there is not enough space on the disk a path refering to temporary directory must be passed to the packing function. Otherwise the database is packed in the same folder.

When the database consists of several main or subbases each one is copied in its own location or to the temporary folder.

```
logical ODABAClient :: PackDatabase (char *cpath, char
            *temp_path )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
cpath	Complete path

The complete path is passed as 0-terminated string with a maximum length of 255 characters.

temp_path	Temporary path The temporary path refers to a folder location for storing temporary files. It is passed as 0-terminated string.

SendClientMessage - Send message to one or all clients

The function sends a message to one or all clients. Depending on the client type the message is displayed on the console (console applications) or as message box.

The message is sent to the client adressed via the client_id. If no client_id is passed (UNDEF) the message is send to all clients except the sending one.

```
logical ODABAClient :: SendClientMessage (int32 client_id, char
                 *mtext, char *mtitle, char mtype )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
client_id	Client the message is send to The client number (client_id) is a number the server has assigned to the client. You can retrieve client numbers by using the GetClientHandle() function which returns client information for all active clients.
mtext	Message text The message text may contain up to 500 characters and must be 0-terminated.
mtitle	Message title When displaying the message in a dialogue box the message title will be displayed in the title bar. The message title should refer in some way to the application the message applys on.
mtype	Message Type

One character indicating the message type can be passed:

I - information

W - Warning

E - Error

All other message types are considered as errors.

SetServerVariable - Set system variable on server side

Systemvariables can be set for the server. This is necessary for controlling functions running on the server side.

Server variables are valid on the server only for the connected client.

```
logical ODABAClient :: SetServerVariable (char *var_name, char
                    *var_string )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
var_name	System variable name Name of the system variable on the server or client side. System variable names must not exceed 40 characters and are provided as 0-terminated strings.
var_string	Value for the system variable The value for a system variable must not exceed 255 characters and is provided as 0-terminated string.

ShutDown - Shut down client

Usually the last ODABAClient handle referring to the client will shut down the client when being destructed. In some cases, e.g. when creating a client with an ini-file and using system services as data catalogue or error logs, some system references are still active and referring to the main client. To be sure that the main client is closed properly you should use the ShutDown() function before destructing the client. Make sure that there are no other references to the client in your application anymore.

The function will delete all resources associated with the client and close the client. When the client is the default or main client, which has been created automatically, the function will close the main client.

```
logical ODABAClient :: ShutDown ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

StartPause - Pause Server

When pausing the server no more transactions can be committed until pausing the server is stopped (StopPause()). The server can pause only after finishing all running transaction commits. If any commit is still running after five minutes or a given number seconds (wait_sec) the server will not pause (error 323). When the server cannot pause the function stops without pausing the server.

The …Pause functions can be used for keeping the database in a consistent state while backing up the database without closing the server. Pause commands should not be used when running long transactions as large imports or database reorganizations.

Transactions will not be committed anymore after pausing the server. The timeout interval for committing transactions is 10 minutes. When not being able to start committing the transaction within the timeout interval the transaction is cancelled.

Any application may access the database in the pause state as long as not writing to the database, i.e. as long as not storing transactions to the database.

For allowing storing data to the database again you must use the StopPause() function.

```
logical ODABAClient :: StartPause (int32 wait_sec )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
wait_sec	Number of seconds to wait The system waits the given number of seconds befor executing the request. Default is 300 seconds (5 minutes).

StatDisplay - Display database statistics

The function creates a database statistic for the database passed in the dbpath. The database must be available via a local or a net drive.

```
logical ODABAClient :: StatDisplay (char *dbpath, char *ppath )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbpath	Complete database path
	The complete database path is passed as 0-terminated string with a maximum length of 255 characters.
ppath	Protocol path
	The protocol path is passed as 0-terminated string. It must point to a valid folder. The file need not exist.

StopPause - Stop pausing server

This command stops pausing the server and allows committing further transactions.

```
void ODABAClient :: StopPause ( )
```

SysInfoDisplay - Display system information

The function creates system information for the database passed in dbpath. The database must be available via a local or a net drive.

```
logical ODABAClient :: SysInfoDisplay (char *dbpath, char *ppath
                   )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbpath	Complete database path
	The complete database path is passed as 0-terminated string with a maximum length of 255 characters.
ppath	Protocol path
	The protocol path is passed as 0-terminated string. It must point to a valid folder. The file need not exist.

operator bool - Compare clients

The function compares two ODABA clients and returns true (YES) when the clients are the same. Clients are the same when they have been assigned using the =operator. Clients are not the seme when the are opened separately.

```
NOTYPE ODABAClient :: operator bool ( )
```

Return value

operator= - Assign ODABA client handle

The function assigns the odaba client to another client handle. When a client has been created for the source the client is referenced in the target handle as well. Whne no client is opened for the source handle the target client gandle will be empty as well.

```
ODABAClient &ODABAClient :: operator= (ODABAClient &client_ref )
```

Return value

client_ref

~ODABAClient - Destructor

When destructing the client the client disconnects from the server. When disconnecting the main client (the first client that has been opened in the application) ´services as error messages and data catalogue are disabled (until another client is constructed, which becomes the main client again).

```
                        ODABAClient :: ~ODABAClient ( )
```

ODABAServer - ODABA Server

A ODABA server will manage any number of databases. After creating an ODABA server it can be started and halted using the functions Start() and Stop(). There is no login required for connecting to the server, however, for accessing a database you may have to pass login information to the server. Login-Information must be passed to the CreateClient function. You can overload this function in your application procedure to provide specific login checkings and other services for an application ODABA2 server.

The ODABA-server maintains a list (catalogue) for database files. This catalogue must be stored under server.ini in the ODABA2 installation path. The catalogue section starts with [ODABA-CATALOGUE].

GetCatlgName - Get database name from catalogue

The function returns the database path for a symbolic name in the catalogue (server.ini). If no catalogue entry with the given name is found the function returns the original name.

```
char *ODABAServer :: GetCatlgName (char *sym_nams, char *cpath,
                                   int32 maxlen )
```

Return value	
sym_nams	
cpath	Complete path The complete path is passed as 0-terminated string with a maximum length of 255 characters.
maxlen	Size of output buffer Specifies the length of the buffer, the information should be stored into. The information is truncated if it is longer than the buffer.

ODABAServer - Constructor

The function creates an ODABA server. The INI-file passed to the server contains information about the system databases and the data catalogue. The programme path is used for searching actions and should be provided when other iINI-files or DLLs are to be loaded from this path.

```
ODABAServer :: ODABAServer (char *inipath, char *prog_path )
```

inipath	
prog_path	Programme path This is the path that is usually passed as first argument to the application.

Start - Start server

```
logical ODABAServer :: Start (int16 wPort )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Stop - Stop server

When stopping the server all client connections are closed. You should ensure that no client is active anymore.

```
void ODABAServer :: Stop ( )
```

~ODABAServer - Destructor

When destructing the server it is stopped if not done so far. You should ensure that no client is active anymore.

```
ODABAServer :: ~ODABAServer ( )
```

OperationHandle - Opreartion Handle

Operation handles can be used for executing operations as expressions or function calls. Usually, an operation is associated with a property handle defining the instance that is passed to the operation as calling object.

CheckExpression - Check validity of an expression

The function checks whether the expression passed to the function is syntactically correct (NO) or not (YES, error).

i0

```
logical OperationHandle :: CheckExpression (Dictionary *dictptr,
                    ACObject *obhandle, char *clsnames, char
                    *exprnames, char *impnames )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dictptr	Dictionary handle An opened dictionary handle is passed.
obhandle	Database Object Handle This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.
clsnames	
exprnames	
impnames	

i01

```
logical OperationHandle :: CheckExpression (Dictionary *dictptr,
                    char *expression, ACObject *obhandle, char
                    *clsnames )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dictptr	Dictionary handle An opened dictionary handle is passed.
expression	OQL expression An OQL expression defines a condition according to the OQL syntax. OQL expressions must always terminate with ';'. The OQL-Expression is passed as 0-terminated string.
obhandle	Database Object Handle This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.
clsnames	

Execute - Execute operation

The function executes a predefined opoeration. The result is returned as property handle.

It is possible to pass a property handle as calling object. If no property handle is passed the one that was used for creating the operation is used as calling object. When using amother property handle than the originating one the type of the properties must be identical.

The result of the operation can be provided with the GetResult() function.

ci - Calling expression with fixed property handle

This implementation calls the expression with the property handle passed when the operation has been created. The selection in the property handle may change but not the property handle.

```
logical OperationHandle :: Execute (ParmList *parameters )
```

Return value The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

parameters

i01 - Calling operation with variable property handle

This implementation allows changing the property handle from call to call. After each call the original property handle for the operation will be put into place again, i.e. tha calling object passed is being used only as long as the Execute function runs. Use this implementation with care, since the property handle passed to the function should be of the same type as the one used for creating the operation.

```
logical OperationHandle :: Execute (PropertyHandle &call_object,
                           ParmList *parameters )
```

Return value The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

parameters

GetDimension - Get dimension of returned value

The function returns the dimension for the instance created by the function. If the result dimension can not be determined the function returns -1 (AUTO).

In some cases the dimension for the result can be provided ater executing the expression. In this cate the function also returns -1 (AUTO).

```
int32 OperationHandle :: GetDimension ( )
```

Return value The dimension describes the property dimension. this is the maximum number of instances that can be stored for the property. The function returns 0 (UNDEF) if there is no limit (collection) or the dimension (cardinality) defined for the property.

GetResult - Get result from the operation

The function returns a property handle that contains the result of the last execution of the expression.

```
PropertyHandle &OperationHandle :: GetResult ( )
```

Return value

GetSize - Get size of returned value

The function returns the area size for the instance created by the function. If the result size can not be determined the function returns -1 (AUTO).

```
int32 OperationHandle :: GetSize ( )
```

Return value	Size of the instance or property area.

Open - Open operation handle

To execute operations the opration handle must be opened. After opening the operation handle an expression or a function can be associated with the operation handle for being executed.

```
logical OperationHandle :: Open (PropertyHandle &prophdl_ref )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.

OperationHandle - Constructor

The constructor creates an operation handle with the passed property handle as calling object.

i00

```
                        OperationHandle :: OperationHandle
                  (PropertyHandle &prophdl_ref )
```

prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.

i01

```
                          OperationHandle :: OperationHandle (
                    )
```

ProvideExpression - Create expression definition

The function checks the expression and creates an internal epression presentation.

ci

```
logical OperationHandle :: ProvideExpression (char *expression,
                    ParmList *parameters )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
expression	OQL expression An OQL expression defines a condition according to the OQL syntax. OQL expressions must always terminate with ';'. The OQL-Expression is passed as 0-terminated string.
parameters	

i01

```
logical OperationHandle :: ProvideExpression (DictionaryHandle
                    &dictionary, ACObject *obhandle, char
                    *class_names, char *expr_names, PropertyHandle
                    **parmlist )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dictionary	
obhandle	Database Object Handle This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.
class_names	

expr_names

parmlist

PIREPL - Replace options

This option is used to control copy or duplicate operations for instances. The replace option is based on the existence of in instance in a collection, i.e. whether an instance with the selected sort key of the target collection does already exist in the target collection (local existence) or in one of the base collections of the target collection (global existence).

Usually, when copying referenced instances the replace option is passed to the subsequent copy operations.

REPL_relationships - Copy relationships

The function copies primary relationship collections (not having been mared as secondary). Together with 'instance' it has the same effect as 'all'.

REPL_instance - Copying parts owned by the instance

This option is used to copy attributes and all linked instances that are owned by the instance. Relationships are not copied.

PIStack - Property handle stack

A property handle stack allows defining a series of related property handles. A Property handle stack can be defined for a property handle and allows activating a new and saving the current handle using the Push() function and re-activating the previous handle using the Pop() function. Thus, it becomes possible, e.g. defining a sequence of subsequent selections with the possibility of going back to the prevoius level.

PropertyHandle - Property Handle

Property handle are used to handle persistent or transient **data source**. A data source is a collection, object instance or an elementary database field. A data source contains the data for a property of a specific object.

A property handle usually handles a collection of subsequent object instance. In special cases the collection is singular (e.g. the 'direction' for a persion is exactly one 'Adress' object instance). In other cases the instance is elementary (as eg the given names of a person).

A property handle has a cursor function that allows to select one of the instances in the collection as the "current" instance. Only from the selected instance you can retrieve data by means of subsequent property handles or Get-functions (GetString(), GetTime(), ...) for elementary datasources.

Generic Property handles

You can define generic property handles using the generic property handle contructor (PH(type)()). This requires that you have created a C++ header file for the referenced type. In this case you can access elementary data field in the instance directly referring to the generated class members. For references the instance contains corresponding generic property handles that you can reference by class member name as well. In this case you need not to create the property handle you want to access. This makes programming simpler but in this case you must recompile the application when changing the database structure. This is not necessary when referring to property handles hierarchies created in the appplication.

Property handle hierarchies

Property handles form a tree that defines a specific view in an application. When defining this view once the property handles cann be used as long as the application follows the defined view. When defining a property handle for "AllPersons", which is an extent in the database, you can define sub-ordinated property handles for 'name', 'children', and 'company', which refer to the persons name, its children and the company the person is working for. When selecting another person in the AllPerson property handle the datasources for 'name', 'children' and 'company' will change. This, however, is maintained automatically by the systen, i.e. when changing the selection in an upper property handle the data

Add - Add instance to collection

You can add instances to any type of collection or reference. When adding an instance to a collection the cardinality for the collection is checked as well as unique key reuirements. When adding an instance to an owning collection or reference a new database instance will be created. Adding an instanc to a collection or reference is possible by position, key (when sort orders are defined for the collection) or with an initialising instance. When terminating successfully the instance added to the collection is selected in the property handle.

The position passed (set_pos0) when adding an instance to a collection has an effect only, when the collection is unordered. Otherwise the position is determined by the key value. For ordered instances you must always pass a sort or ident key value. When passing a sort key this must correspond to the active order (index) set for the property handle. When an __AUTOIDENT key has been defined the next instance number is determined in the collection or in its most top super set (based collections).

{b Collection based references}

When adding an instance to a collection based reference or collection (ie a subset of another collection) the function checks whether an instance with the same key (ident key of the instance to be added) does already exist in the base collection (super set). In this case no new instance is created but the instnce found in the base collection is added to the current collection or reference. The instance values are updated from the values of the instance found in the base collection. When no instance has been found the new instance (key) is added to the base collection and than to the current collection. In any case the instance is shared between the base and the current collection, i.e. both refer to the same instance.

For base collections you must always provide an ident key. The function will store the passed keys in an initial instance (which can also be passed by the application). From this initial instance the function extracts the ident key for searching the instance in the basic collections, i.e. the ident key is the minimum information an instance should have.

{b Using shared base structure instances}

When the collection refers to an instance which has one or more shared base instances the instance must con-

i0 - Add instance

This function is used to insert an instance at a certain position in an unordered collection. The function inserts the instance passed at the passed position (set_pos_w). When no position or AUTO is passed as position the instance is inserted infront of the selected instance. When no instance is selected the new instance is appended to the end of the collection.

When the collection is ordered the position passed will be ignored and the instance will be inserted in the collection according to the key passed within the instance.

```
Instance PropertyHandle :: Add (Instance newinst, int32
                          set_pos0_w )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
newinst	New instance The new instance refers to the data area of the instance to be added to a collection. The instance contains a reference to a propertly structured area. You can pass the instance as (void *) which will be automatically converted into an instance handle. Only attributes of the new instance are added to the database. References or relationships in the new instance will be ignored (if there are any).
set_pos0_w	Position in collection

The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance.

For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored.

Special positions are

CUR_INSTANCE

CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance.

FIRST_INSTANCE

FIRST_INSTANCE (0) refers to the first instance in a collection according to the selected index (sort order).

LAST_INSTANCE

FIRST_INSTANCE (0) refers to the last instance in a collection according to the selected index (sort order).

i04 - Add instance by property value

The function checks whether the property handle passes a numerical value or not. When passing a numerical value the function creates an instance at the position according to the number passed in the property handle (-> "Create instance at position"). Otherwise the value in the property handle is interpreted as string key, which will be converted into key and adds an instance by key value to the collection (-> "Add instance by key value")..

```
Instance PropertyHandle :: Add (PropertyHandle &prop_hdl )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i1 - Create instance at position

This function is used to create an instance at a certain position in an unordered collection. The function creates the instance at the passed position (set_pos_w). When AUTO is passed the instance is created infront of the selected instance. When no instance is selected the new instance is appended to the end of the collection. The instance in the property handle can be initialized before calling the Add function calling GetInitInstance() and setting initial property values. In this case the init_inst option must be set to YES when calling the function (otherwuise the initialized instance will be ignored) and the function operates similar to the "Add instance at position" function. The instance can be also initialized before adding to the collection using the DBInitialized event in the structure context.

When the collection is ordered the position passed will be ignored and the instance will be inserted in the collection according to the key passed within the initialized instance. When the instance is not initialized, the instance is created with the default values defined in the data model (structure definition).

```
Instance PropertyHandle :: Add (int32 set_pos0_w, logical in-
                    it_inst )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
set_pos0_w	Position in collection

The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance.

For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored.

Special positions are

CUR_INSTANCE

CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance.

FIRST_INSTANCE

FIRST_INSTANCE (0) refers to the first instance in a collection according to the selected index (sort order).

LAST_INSTANCE

FIRST_INSTANCE (0) refers to the last instance in a collection according to the selected index (sort order).

init_inst

Add initialized instance

The option forces the function to use the internal instance area for creating a new instance in the collection. This instance has been provided using the GetInitInstance() function and has to be filled by the application befor calling the Add() function.

Default:YES

i2 - Add instance by key

The function adds an instance by key (sortkey). When the collection is unordered the sort key will be interpreted as ident key (when no extra ident key is passed). When a sort order has been selected for the collection that is not the ident key and that does not contain the ident key, both, sortkey and identkey must be passed to the function.

Instead of the key you may pass an instance that contains the values for the keys. This can solve problems when having several unique key indexes for a collection. You can also use the GetInitInstance() function to set the initial values in the instance area of the property handle. In this case init_inst must be set to YES.

```
Instance PropertyHandle :: Add (Key sortkey, Key identkey_w, In-
                          stance newinst_w, logical init_inst )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area.
	You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
sortkey	Sort key value
	The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the ({.r pib.StrinToKey}()) function. Regardles on the type key values are passed as (char *) areas.
identkey_w	Ident key value
	The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the ({.r pib.StringToKey}()) function. Regardles on the type key values are passed as (char *) areas. NULL indicates that no key value has passed.
newinst_w	New instance

The new instance refers to the data area of the instance to be added to a collection. The instance contains a reference to a propertly structured area.

You can pass the instance as (void *) which will be automatically converted into an instance handle. Only attributes of the new instance are added to the database. References or relationships in the new instance will be ignored (if there are any).

Default: Instance() (empty instance)

init_inst	Add initialized instance The option forces the function to use the internal instance area for creating a new instance in the collection. This instance has been provided using the GetInitInstance() function and has to be filled by the application befor calling the Add() function. Default:YES

i3 - Add instance by index and key

This function can be used to add an instance to different collections. The collection referenced in the property handle might be unordered, which requires adding an instance by pposition. A base collection defined for the collection, however, requires a key for adding an instance. In this case, the position, a sortkey and an identkey might be necessary. Instead of the key values an instance (newinst_w) can be passed or the velues can be set in the instance area of the property handle (GetInitInstance()). You may also use the DBInitialized event handler for settinig initial values for the instance to be created.

```
Instance PropertyHandle :: Add (int32 set_pos0, Key sortkey, Key
                   identkey_w, Instance newinst_w, logical in-
                   it_inst )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
set_pos0	Position in collection

The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance.

For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored.

Special positions are

CUR_INSTANCE

CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance.

FIRST_INSTANCE

FIRST_INSATNCE (0) refers to the first instance in a collection according to the selected index (sort order).

LAST_INSTANCE

FIRST_INSATNCE (0) refers to the last instance in a collection according to the selected index (sort order).

sortkey

Sort key value

The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the ({.r pib.StrinToKey}()) function. Regardles on the type key values are passed as (char *) areas.

identkey_w

Ident key value

The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the ({.r pib.StringToKey}()) function. Regardles on the type key values are passed as (char *) areas. NULL indicates that no key value has passed.

newinst_w

New instance

	The new instance refers to the data area of the instance to be added to a collection. The instance contains a reference to a propertly structured area.
	You can pass the instance as (void *) which will be automatically converted into an instance handle. Only attributes of the new instance are added to the database. References or relationships in the new instance will be ignored (if there are any).
	Default: Instance() (empty instance)
init_inst	Add initialized instance
	The option forces the function to use the internal instance area for creating a new instance in the collection. This instance has been provided using the GetInitInstance() function and has to be filled by the application befor calling the Add() function.
	Default:YES

AddReference - Add persistent instance

The function adds an instance selected in another property handle to the collection/reference of the current property handle. Both property handles must have the same type or the same base type if they are weak typed. You can use AddReference() only for not owning collections. Usually the function is used to fill temporary extents with instances, which have been selected for special purposes.

You can only add instances that are defined in the same database as the instances of the target propüerty handle. It is also not possible to add an instance by reference from a server database to a local database or vize versa.

After addin the instance to the property handle the added instance is the currently selected one.

Events

When adding an instance to a collection the function fires an i Insert-event. You can use the insert event for checking the operation and deny it. After the instance has been added an i Inserted-Event is generated.

When the post event handler has been modifying the instance an additional i Stored-event might be created. When selecting the instance in the property handle a final i Read-event is generated..

```
logical PropertyHandle :: AddReference (PropertyHandle
                 &source_handle, int32 set_pos0_w )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
source_handle	Source property handle

The source property handle must be opened and an instance must be selected in the handle.

set_pos0_w	Position in collection

The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance.

For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored.

Special positions are

CUR_INSTANCE

CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance.

FIRST_INSTANCE

FIRST_INSTANCE (0) refers to the first instance in a collection according to the selected index (sort order).

LAST_INSTANCE

FIRST_INSTANCE (0) refers to the last instance in a collection according to the selected index (sort order).

AllocDescription - Allocate property description

The function creates an empty description for the property handle. When the property handle is already associated with a description this will be removed from the property handle.

i0

```
logical PropertyHandle :: AllocDescription ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

i1

```
logical PropertyHandle :: AllocDescription (DBHandle *dbhandle,
                    char *fldnames, char *fldtypes, SDB_RLEV
                    ref_level, uint16 size, uint16 precision,
                    uint16 dimension )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbhandle	Pointer to database handle This is pointer to an opened database handle. The database handle can be an opened database handle DatabaseHandle as well as an opened dictionary handle (DictionaryHandle).
fldnames	
fldtypes	
ref_level	Reference level The reference level describes the way and the level of instance references.
size	Size Size of the instance or property area.
precision	Precision The precision defines the number of decimal positions behind the decimal point for numerical valued. For date and time values it defines the way of presenting the values in charachter presentations.
dimension	Dimension The dimension describes the property dimension. this is the maximum number of instances that can be stored for the property. The function returns 0 (UNDEF) if there is no limit (collection) or the dimension (cardinality) defined for the property.

AllocateArea - Allocate instance area

The function allocates a data area for a property handle if this has not yet been done. If the property handle is linked to the data area of another property handle this link is deleted an a private data area is allocated.

You can use this function when you create dummy property handles which are controlled by the application.

```
logical PropertyHandle :: AllocateArea ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Cancel - Cancel selection

The function resets modifications made on the internal instance and cancels the selection. After cancelling an instance the property handle has no current selection. All subordinated property handles are cancelled as well.

```
logical PropertyHandle :: Cancel ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

CancelBuffer - Cancel all buffered instances

The function will release all instances in a buffer. The next Get() access will fill the buffer again.

```
logical PropertyHandle :: CancelBuffer ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ChangeBuffer - Change collection buffer count

Instances for a PropertyHandle can be read in block mode. The function allocates a buffer size to the property handle, i.e. the number of instances that will be read at once. When a buffer has already been allocated you can use the function to change the buffer size. Passing 0 or 1 will change from buffered read to unbuffered access (same as ReleaseBuffer()).

Buffered access can be used for reading collections with fixed type, only. Trying to enable buffered access for untyped or weak typed collections will fail without writing an error to the error log file.

When passing AUTO as buffer size the number of instances stored in the collection is reserved. Thus, you will read the complete collection into the buffer.

When a filter has been defined for the property handle only instances are read into the buffer that fulfill the filter condition.

Usually the buffer is filled automatically when reading an instance that is not contained in the buffer. The system tries to read as many instances as defined in buffer size from the current position. Subsequent Get() request will read from the buffer as long as possible. Cancel() will cansel the current selection but not the instances in the buffer.

To position the buffer on a certain instance you can use the ReadBuffer() function. For resetting the buffer you can use the CancelBuffer() function.

You cannot use blockmode for views containing references. In this case the function will ignore the request and buffer size remains 1. There are also problems in client/server mode when referring to sub-property handles for references in instances that heve been reading in blockmode. Moreover, blockmode cannot be used for updating instances.

```
int16 PropertyHandle :: ChangeBuffer (int16 buffnum )
```

Return value	This is the number of instance buffers allocated to the collection handle.
buffnum	Number of instance buffers

This is the number of instance buffers allocated to the collection handle.

ChangeMode - Change access mode

The function allows changing the access mode for a PropertyHandle. This requires that the batabase object access mode is higher or equal to the mode that is going to be activated.

You can always change from higher modes (PI_Write, PI_Update) to lower modes (PI_Read). Changing from read to update or write mode has, however, some limitations. You may change the mode in any directions for extent property handles and for property handles referring to updateable relationships. Changing the mode for non updateable relationschips or references, however, is possible only to PI_Read or PI_Write. Changing to PI_Write is possible only, when the parent PropertyHandle is opened with or set to PI_Write.

Changing the access mode will cancel all subsequent PropertyHandles and change the mode to the mode requested for teh current PropertyHandle.

Property handle running in block mode can be accessed in read mode, only. Change mode will fail when attemtimg to change the mode to update or write in this case.

```
PIACC PropertyHandle :: ChangeMode (PIACC newmode )
```

Return value	Access mode that was valid for the property handle.
newmode	New access mode Access mode to be set for the property handle.

Check - Check property handle

The function checks whether a property handle is valid and opened.

Usually property handles are opened on demand, i.e. when access to any database source is required. After opening a property handle a structure definition is available and access is possible in principle.

If the property handle is invalid or not opened the function returns YES, otherwise NO.

```
logical PropertyHandle :: Check (logical ini_opt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
ini_opt	Initialize option The option forces the function to initialize the property handle if not yet done. Usually property handles are initialized automatically on demand, i.e. when being used the first time. Default: NO

CheckWProtect - Is current instance permanent write protected?

Instances can be marked as persistent write protected (SetWProtect()). Such instances cannot be updated by any user. The function returns the persistent write protection state.

Persistent write protection can be reset with ResetWProtect().

```
logical PropertyHandle :: CheckWProtect ( )
```

Return value	Instances can be write protected because they are used by anoter application (pessimistic locking) or because they are persistent wirite protected. 0 - instance is not write protected 1 - instance is temporarily write protected 2 - instance is persistent write protected

Close - Close Property Handle

The function will close the property handle without destroying it. The handle can be re-opend later again.

When closing or destroying the property handle unsaved modifications will be saved atomatically. This might cause a number of activities including event handlers.

Since the close fonction or the destructor are called implicitely in many cases, the function will push the error and restore it, when no error has detected while closing the property handle. When terminating normally, the error set is the same as before calling the close function. Otherwise the error returned in SDBError or CTXError is the error from the close function.

```
logical PropertyHandle :: Close ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Compare - Compare the values for two property handles

The function compares the values of the two property handles. The function cannot compare collections.

The function retuns -1 when the the value for the property handle is lower than the value for the passed property handle. The function retuns 1 when the the value for the property handle is higher than the value for the passed property handle. The function returns 0 if the values are equal.

The function returns ERIC (-99) if the values are not compareble, i.e. no instance selected for the property or invalid property handle.

i00 - Compare with other property handle

This implementation compares the value in the property handle with the value in the passed property handle. Data conversion is performed when required.

```
int16 PropertyHandle :: Compare (const PropertyHandle &cprop_hdl
                    )
```

Return value	The result of a comarision is an integer value with the following meaning: 0 - both operands have the same value 1 - the calling operand is greater than the passed operand -1 - the calling operand is smaller than the passed operand
cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i01 - Compare with string value

This implementation compares the value in the property handle with the value in the passed string. Data conversion is performed for the string when required.

```
int16 PropertyHandle :: Compare (char *string )
```

Return value	The result of a comarision is an integer value with the following meaning: 0 - both operands have the same value 1 - the calling operand is greater than the passed operand -1 - the calling operand is smaller than the passed operand
string	String area Pointer to the 0-terminated string area.

i02 - Compare with 32-bit integer value

This implementation compares the value in the property handle with the passed integer value. Data conversion is performed for the passed value when required.

```
int16 PropertyHandle :: Compare (int32 long_val )
```

Return value	The result of a comarision is an integer value with the following meaning: 0 - both operands have the same value 1 - the calling operand is greater than the passed operand -1 - the calling operand is smaller than the passed operand
long_val	Integer value The value is passed as platform independent 32-bit integer value.

i03 - Compare with double value

This implementation compares the value in the property handle with the passed double float value. Data conversion is performed for the passed value when required.

```
int16 PropertyHandle :: Compare (double double_val )
```

Return value	The result of a comarision is an integer value with the following meaning: 0 - both operands have the same value 1 - the calling operand is greater than the passed operand -1 - the calling operand is smaller than the passed operand
double_val	

i04 - Compare with date value

This implementation compares the value in the property handle with the passed date value. Data conversion is performed for the passed value when required. Converting date values to string values may result in different string values for the same date value depending on the national setting. Hence, string values should not be compared with date values. In this case it is more appropriate to compare the date values directly (ph.GetDate() == date_val).

```
int16 PropertyHandle :: Compare (dbdt date_val )
```

Return value	The result of a comarision is an integer value with the following meaning: 0 - both operands have the same value 1 - the calling operand is greater than the passed operand -1 - the calling operand is smaller than the passed operand
date_val	Date value The data value is passed in the internal data format.

i05 - Compare with time value

This implementation compares the value in the property handle with the passed time value. Data conversion is performed for the passed value when required. Converting time values to string values may result in different string values for the same time value depending on the national setting. Hence, string values should not be compared with time values. In this case it is more appropriate to compare the time values directly (ph.GetTime() == time_val).

```
int16 PropertyHandle :: Compare (dbtm time_val )
```

Return value	The result of a comarision is an integer value with the following meaning: 0 - both operands have the same value 1 - the calling operand is greater than the passed operand -1 - the calling operand is smaller than the passed operand
time_val	Time value The time value is passed in the internal data format.

CompareKey - Compare two ident key values

The function compares two ident key values. The keys are compared according to the data types of it's components.

The function retuns -1 when the the value for the first key (ident_key1) is lower than the value for the second key (ident_key2). The function retuns 1 when the the first value is higher than the second one. The function returns 0 when the keys are are equal.

The function returns ERIC (-99) if the values are not compareble, i.e. when no ident key has been defined or when the property handle is invalid.

```
int8 PropertyHandle :: CompareKey (Key ident_key1, Key
                       ident_key2 )
```

Return value	Compare functions return the folowing values:
	0 - Both values are equal
	1 - The second value is larger than the first
	-1 - The second value is smaller than the first
ident_key1	First ident key value
	The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey() function. Regardles on the type key values are passed as Key handle or (char *) areas.
ident_key2	Second key value
	The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey() function. Regardles on the type key values are passed as Key handle or (char *) areas.

CompareSortKey - Compare keys accroding to current sort order

The function compares two key values for the selected sort key (SetOrder()). The keys are compared according to the data types of it's components.

The function retuns -1 when the the value for the first key (sort_key1) is lower than the value for the second key (sort_key2). The function retuns 1 when the the first value is higher than the second one. The function returns 0 when the keys are are equal.

The function returns ERIC (-99) if the values are not compareble, i.e. when the collection is unordered or when the property handle is invalid.

```
int8 PropertyHandle :: CompareSortKey (Key sort_key1, Key
                                        sort_key2 )
```

Return value	Compare functions return the folowing values: 0 - Both values are equal 1 - The second value is larger than the first -1 - The second value is smaller than the first
sort_key1	First sort key The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey() function. Regardles on the type key values are passed as Key handle or (char *) areas.
sort_key2	Second key The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey() function. Regardles on the type key values are passed as Key handle or (char *) areas.

CompareType - Check properties for comparability

The function checks the comparability for the data of two property handles. Usually property handles are considered as comparable when they have the same type. When requesting data convertion (passing YES for convert) the function checks, whether the types are comparable after conversion.

```
logical PropertyHandle :: CompareType (PropertyHandle &prop_hdl,
                          logical convert )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.
convert	Conversion option To allow data conversion the value must be set to YES. To supress data conversion NO shold be passed.

Contains - Does property contain text

The function checks, wether the instance for the selected property handle contains the text passed in reg_string. The function returns true (YES), when the text has been found.

This version supports simple string expressions as 'string'', '*string', 'string*' and '*string*'. When not beginning or terminting the search string with an '*', the text must be at the beginning or at the end of a word. Searching for 'string' will search for whole words, only.

i00 - Search for text in instance

When the instance is a structures instance, the function is called for all attributes and MEMO fields in the selected instance. Otherwise the data in the attribute or MEMO field is checked.

```
logical PropertyHandle :: Contains (char *reg_string, logical
                    case_opt )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
reg_string	Regular string expression
	The string contains a regular string expression passed as 0-terminated string.
case_opt	Case sensitive
	The option indicates case sensitive data in text (YES)

i01 - Search for text in property

The function searches for the text in the property passed in 'prop_path'. This implementation has been provided for convinience. It does the same as prop_path.Contains(reg_string).

```
logical PropertyHandle :: Contains (char *reg_string, char
                    *prop_path, logical case_opt )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
reg_string	Regular string expression
	The string contains a regular string expression passed as 0-terminated string.
prop_path	Property path
	The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.
case_opt	Case sensitive
	The option indicates case sensitive data in text (YES)

ConvertToWinChar - Converts ASCII character into Windows compatible ANSI character set

The function converts ASCII character in a text property handle into Windows compatible ANSI character. The data is updated in-place and a data modification is signaled.

```
logical PropertyHandle :: ConvertToWinChar ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Copy - Copy instance

The function allows copying instances from one collection into another. The function creates a new instance that is filled with the data from the source instance. The function allows copying instances between collections in different databases.

The source handle must point to a selected instance that is copied into the target handle collection. The target property handle must be opened for write access.

Source and target property handle may refer to collections of different types. Properties of the copied instances are copied by name. Names of property handles in base structured are considered without prefix. When the target collection is weak typed the type for the target instance must be set by the application before copying the instance (SetType()).

When the instance to be copied does already exist in the target collection, which is determined according to the sort order selected for the target collection, different replace options ({r. PIREPL}) will control the copy behaviour (replopt). Copying an existing instance without requesting replacement of existing instances will result in an error. When no unique sort order has been selected instances are considered as not existing and are copied always.

Copying instances includes copying related collections (references and relationships). Only primary relationships (not defined as secondary) are copied to avoid unlimited recursions. When copyinf related collection the replace option is passed to the subsequent copy operations.

The copy type supports a two phase copy process. Normally copying an instance owning parts and primary relationships are copied at once. Sometimes this may lead to logical problems since it is not always possible to ensure that base collections are copied before copying primary relationships based on it. Since this leads to incomplete copy operations a two phase copy can be used copying first the owning parts (REPL_instance) and later with a second call of the copy function the primary relationships (REPL_relationship).

When Copy terminates succesfully the copied instance is selected in the target property handle and the instance is returned. When terminating with errors the function returns an emty instance.

i00 - Copy selected instance

The function copies the selected instance from the source property handle to the target property handle.

```
Instance PropertyHandle :: Copy (PropertyHandle &source_handle,
                                 PIREPL replopt, int16 copy_type )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
source_handle	Source property handle The source property handle must be opened and an instance must be selected in the handle.
replopt	Replace option The replace option controls the behaviour of the copy function. Options that can be used here are: REPL_none - do not replace existing instances REPL_direct - copy attributes, only (but no global identities) REPL_GUID - copy attributes including global identity REPL_local - replace collections owned by the instance REPL_all - replace primary relationships REPL_no_create - copy primary relationships without creating new instances
copy_type	Copy type The copy type determins the way of copying instances: REPL_all - copies all instances recursively owned by the instance and the primary relationships REPL_Instance - copies all instances recursively owned by the instance REPL_relationship - copies the primary relationships

i01 - Copy and rename

The implementation copies the selected instance to the target property handle. The sort key of the instance according to the settings in the target property handle is changed to the passed key value. When the new key value does already exist in the target collection and the collectin index requires unique keys, no instance will be copied and the function returns NULL.

```
Instance PropertyHandle :: Copy (PropertyHandle &source_handle,
                    Key new_key, PIREPL replopt, int16 copy_type )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
source_handle	Source property handle The source property handle must be opened and an instance must be selected in the handle.
new_key	New key for the instance The key passed for renaming the instance must be structured according to the currently selected sort order.
replopt	Replace option The replace option controls the behaviour of the copy function. Options that can be used here are: REPL_none - do not replace existing instances REPL_direct - copy attributes, only (but no global identities) REPL_GUID - copy attributes including global identity REPL_local - replace collections owned by the instance REPL_all - replace primary relationships REPL_no_create - copy primary relationships without creating new instances
copy_type	Copy type

The copy type determins the way of copying instances:

REPL_all - copies all instances recursively owned by the instance and the primary relationships

REPL_Instance - copies all instances recursively owned by the instance

REPL_relationship - copies the primary relationships

i02 - Copy to position

This implementation copies the selected instance from the source instance at the position passed in set_pos_w in the target collection. When the passed position is greater that the collection count, the position is changed to the number of instances in the target collection. When passinf AUTO the instance is positioned infront of the selected instance in the target collection (if an instance is selected) or appended at the end of the list.

When the collection is ordered, the position passed is ignored and the instance is inserted according to the key value in the source instance.

```
Instance PropertyHandle :: Copy (PropertyHandle &source_handle,
                          int32 set_pos0, PIREPL replopt, int16
                          copy_type )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
source_handle	Source property handle The source property handle must be opened and an instance must be selected in the handle.
set_pos0	Position in collection

The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance.

For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored.

Special positions are

CUR_INSTANCE

CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance.

FIRST_INSTANCE

FIRST_INSATNCE (0) refers to the first instance in a collection according to the selected index (sort order).

LAST_INSTANCE

FIRST_INSATNCE (0) refers to the last instance in a collection according to the selected index (sort order).

replopt

Replace option

The replace option controls the behaviour of the copy function. Options that can be used here are:

REPL_none - do not replace existing instances

REPL_direct - copy attributes, only (but no global identities)

REPL_GUID - copy attributes including global identity

REPL_local - replace collections owned by the instance

REPL_all - replace primary relationships

REPL_no_create - copy primary relationships without creating new instances

copy_type

Copy type

The copy type determins the way of copying instances:

REPL_all - copies all instances recursively owned by the instance and the primary relationships

REPL_Instance - copies all instances recursively owned by the instance

REPL_relationship - copies the primary relationships

i03

```
Instance PropertyHandle :: Copy (PropertyHandle &source_handle,
                 Key new_key, int32 set_pos0, PIREPL replopt,
                 int16 copy_type )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
source_handle	Source property handle The source property handle must be opened and an instance must be selected in the handle.
new_key	New key for the instance The key passed for renaming the instance must be structured according to the currently selected sort order.
set_pos0	Position in collection The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance. For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored. Special positions are **CUR_INSTANCE** CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance. **FIRST_INSTANCE** FIRST_INSATNCE (0) refers to the first instance in a collection according to the selected index (sort order). **LAST_INSTANCE** FIRST_INSATNCE (0) refers to the last instance in a collection according to the selected index (sort order).
replopt	Replace option

	The replace option controls the behaviour of the copy function. Options that can be used here are:
	REPL_none - do not replace existing instances
	REPL_direct - copy attributes, only (but no global identities)
	REPL_GUID - copy attributes including global identity
	REPL_local - replace collections owned by the instance
	REPL_all - replace primary relationships
	REPL_no_create - copy primary relationships without creating new instances
copy_type	Copy type
	The copy type determins the way of copying instances:
	REPL_all - copies all instances recursively owned by the instance and the primary relationships
	REPL_Instance - copies all instances recursively owned by the instance
	REPL_relationship - copies the primary relationships

CopyData - Copy data from an instance area

The function copies the data from the instance area passed into the selected instance of the property handle. The structure of the instance area must correspond to the structure of the property handle.

Passing 'saveopt' YES will store the instance immediately.

```
logical PropertyHandle :: CopyData (char *instance, logical
                          switchopt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
instance	Instance area

Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area.

You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

switchopt

Unselct option

The option forces the function to unselect the selected instance in the property handle after terminating the function.

CopyDescription - Create a copy for the property description

The function creates a copy for the property definition that can be modified in the application. Do never modify the description profided by the system.

```
logical PropertyHandle :: CopyDescription (DBFieldDef *prop_def
                 )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_def	Property definition The property defintion contains the metadata for the referenced property instance.

CopyHandle - Create a copy of the property handle

The function creates a copy of the property handle when being called with a property handle reference. A copy of a property handle has its own cursor but refers to the same data source (collection, instance, value).

When passing a property handle pointer the function creates another property handle that shares the data source and the cursor with its origin. In this case changing the selection in the origin or the copy will always affect the other handle as well.

i0

```
logical PropertyHandle :: CopyHandle (PropertyHandle &prop_hdl )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i01

```
logical PropertyHandle :: CopyHandle (PropertyHandle
                    *property_handle )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
property_handle	Pointer to a property handle Is a pointer to an (usually) opened property handle.

CopyInst - Copy transient instance

The function allows copying a transient instance into the the collection referenced by the property handle. Source and target may have different structure definitions. Properties are copied by property names. Type conversion is performed when necessary. After copying the attributes the function tries to locate the instance according to the sort key or the ident key (if no order is defined). Otherwise the function tries to locate the instance in the base collection (if there is any). When the instance does already exist in the collection it will be overwritten according to the replace options. Otherwise the instance is added to the collection.

```
Instance PropertyHandle :: CopyInst (char *srceinst, smcb
                    *srcesmcb, PIREPL replopt, int16 copy_type )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
srceinst	Source instance

	A pointer to an instance of the defined type must be passed. The instance can be a persistent instance read from another location or a transient one. The structure of the instance must confirm to the passed structure definition (srcesmcb). References or relationships in the new instance will be ignored (if there are any).
srcesmcb	Pointer to general structure definition The smcb is a more general way to define structure (DBStructDef). It contains information for the structure and its properties. The definition describes the structure of the instance passed to the function.
replopt	Replace option The replace option controls the behaviour of the copy function. Options that can be used here are: REPL_none - do not replace existing instances REPL_direct - copy attributes, only (but no global identities) REPL_GUID - copy attributes including global identity REPL_local - replace collections owned by the instance REPL_all - replace primary relationships REPL_no_create - copy primary relationships without creating new instances
copy_type	Copy type The copy type determins the way of copying instances: REPL_all - copies all instances recursively owned by the instance and the primary relationships REPL_Instance - copies all instances recursively owned by the instance REPL_relationship - copies the primary relationships

CopySet - Copy collection

The function copies all instances of the source collection to the target collection. The function works well for multiple references as well as for single references. The function cannot be used for MEMO fields.

The target Property Handle must be opened in unpdate mode (PI_Write).

Source and target need not to refer to the same object type. Attributes and references are copied by property name, i.e. they are assigned by looking for the same property name in the target instance. Data conversin is performed automatically if possible. This includes also convertind imbedded instances into references and reverse.

The target collection is not emptied automatically. Existing instances in the target are replaced according to the replace option passed to the function.

When successfull the function returns the number of instances copied to the target collection (including 0 when the source was empty). When the function has terminated because of an error it returns AUTO (-1). Source and target handle are not positioned after terminating the function.

```
int16 PropertyHandle :: CopySet (const PropertyHandle
                  &csource_handle, PIREPL replopt, int16
                  copy_type )
```

Return value	
csource_handle	Source property handle The source property handle must be opened and an instance must be selected in the handle.
replopt	Replace option

The replace option controls the behaviour of the copy function. Options that can be used here are:

REPL_none - do not replace existing instances

REPL_direct - copy attributes, only (but no global identities)

REPL_GUID - copy attributes including global identity

REPL_local - replace collections owned by the instance

REPL_all - replace primary relationships

REPL_no_create - copy primary relationships without creating new instances

copy_type	Copy type

The copy type determins the way of copying instances:

REPL_all - copies all instances recursively owned by the instance and the primary relationships

REPL_Instance - copies all instances recursively owned by the instance

REPL_relationship - copies the primary relationships

CreateTempExtent - Creates a temporary extent

This function can be used to open a property handle for a transient collection property (transient reference). The property must have a definition that has been provided by the constructor or an appropriate open function.

The function creates a temporary collection with the structure of the transient reference property. The property handle is opened in write mode, always. If the collection is already opened the function returns without error. You can reset the transient reference using the ResetTransientProperty() function.

The function returns an error (YES) if the property handle does not define a transient reference.

i0

```
logical PropertyHandle :: CreateTempExtent (ACObject *obhandle )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
obhandle	Database Object Handle This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.

i01

```
logical PropertyHandle :: CreateTempExtent (PropertyHandle
                    &prophdl_ref, char *extnames_w )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.
extnames_w	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

Delete - Delete/remove instance from collection

The function removes the selected instance from the collection. Whe passing a key or a position the function selects the instance with the passed key or the given position before removing it from the collection. Deleting by key is possible only when a sort order with a unique key has been selected. For removing an instance from a collection the property handle must be opened in write mode. For deleting the instance the selected instance must be available in write mode as well.

Removing an instance from a collection will delete the instance as well, when the collection is owning the instance or when the instances in the collection do depend on the collection. You may also request deleting the instance by passing YES for the del_dep parameter.

Removing an instance from a collection it will maintain automatically inverse references. Moreover, the instance is removed from all derived collections (subsets). This means the instances might be deleted also, when being dependent on one of the derived collections.

When deleting the instance derived instances are deleted as well. All inverse relationships and subset relationships are maintained.

i01 - Delete instance by key

The function deletes an instance by key. When no instance with the passed key can be located, the function terminates with error.

```
logical PropertyHandle :: Delete (Key sort_key )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
sort_key	Sort key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey}() function. When no key is passed by the application an empty key (without data area) will be passed.

i02 - Delete instance by position

The function deletes the instance at the position passed to the function (set_pos0_w). When AUTO is passed the current instance is deleted. Dependent instances are usually deleted as well (del_dep=YES).

When passing NO, depending instances are not deleted, as long as they are not owned by the current instance.

```
logical PropertyHandle :: Delete (int32 set_pos0_w, logical
                 del_dep )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
set_pos0_w	Position in collection The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance. For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored. Special positions are **CUR_INSTANCE** CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance. **FIRST_INSTANCE** FIRST_INSTANCE (0) refers to the first instance in a collection according to the selected index (sort order). **LAST_INSTANCE** FIRST_INSTANCE (0) refers to the last instance in a collection according to the selected index (sort order).
del_dep	Delete dependent instances

Usually this option is set to YES, i.e. all dependent instances are deleted when being removed from the collection. Passing NO dependent instances are not deleted but removed from the collection, only.

Instances owned by the collection are, however, always deleted.

i03

```
logical PropertyHandle :: Delete (PropertyHandle &prop_hdl )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

DeleteSet - Delete/remove all instance in a collection

The function removes all instances from the collection. Instances are removed/deleted as described in the Delete() function. When the finction fails the collection remains unchanged.

```
logical PropertyHandle :: DeleteSet (logical del_dep )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
del_dep	Delete dependent instances Usually this option is set to YES, i.e. all dependent instances are deleted when being removed from the collection. Passing NO dependent instances are not deleted but removed from the collection, only. Instances owned by the collection are, however, always deleted.

Dereference - Dereference collection handle

The function returns a dereferenced property handle. The Area is shared with the Area of the original handle and the instance is associated with the collection of the original handle.

```
PropertyHandle *PropertyHandle :: Dereference ( )
```

Return value	Is a reference to an (usually) opened property handle.

Duplicate - Duplicate instance

You can use duplicate to create a new version of the selected instance in the same collection.

Duplicate should be used only, if the instance to be copied has no shared base instances or if all shared base instances are identified by the selected sort key for the instance. If this is not the case the system trys to create appropriate base instances with refering to the original base instance except base instances identified by __AUTOIDENT or __IDENTITY, i.e. Duplicate() will not copy automatic keys but creates new base instances with new keys.

i01

```
Instance PropertyHandle :: Duplicate (int32 set_pos0, PIREPL
                                replopt )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
set_pos0	Position in collection

The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance.

For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored.

Special positions are

CUR_INSTANCE

CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance.

FIRST_INSTANCE

FIRST_INSATNCE (0) refers to the first instance in a collection according to the selected index (sort order).

LAST_INSTANCE

FIRST_INSATNCE (0) refers to the last instance in a collection according to the selected index (sort order).

replopt	Replace option

The replace option controls the behaviour of the copy function. Options that can be used here are:

REPL_none - do not replace existing instances

REPL_direct - copy attributes, only (but no global identities)

REPL_GUID - copy attributes including global identity

REPL_local - replace collections owned by the instance

REPL_all - replace primary relationships

REPL_no_create - copy primary relationships without creating new instances

i1

```
Instance PropertyHandle :: Duplicate (Key ident_key, PIREPL
                    replopt )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
ident_key	Ident key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey() function.
replopt	Replace option The replace option controls the behaviour of the copy function. Options that can be used here are: REPL_none - do not replace existing instances REPL_direct - copy attributes, only (but no global identities) REPL_GUID - copy attributes including global identity REPL_local - replace collections owned by the instance REPL_all - replace primary relationships REPL_no_create - copy primary relationships without creating new instances

ExecuteInstanceAction - Execute action on instance level

The function calls an action that is defined in the structure context of the current instance. The function is executed on the server side.

The action may use the SetActionResult() function to pass the result of the action to the client application. The result can be retrieved from the client application using the function GetActionResult().

```
logical PropertyHandle :: ExecuteInstanceAction (char
                  *action_name, char *parm_string )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
action_name	Name of the action to be performed

The name of the action is passed as 0-terminated string with a maximum length of 40 significant characters.

parm_string	Parameter string The parameter string is passed as 0-terminated string and contains the parameters according to the conventions of the action called.

ExecutePropertyAction - Execute action on property (collection) level

The function calls an action that is defined in theproperty context of the property handle. The function is executed on the server side.

The action may use the SetActionResult() function to pass the result of the action to the client application. The result can be retrieved from the client application using the function GetActionResult().

```
logical PropertyHandle :: ExecutePropertyAction (char
                 *action_name, char *parm_string )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
action_name	Name of the action to be performed The name of the action is passed as 0-terminated string with a maximum length of 40 significant characters.
parm_string	Parameter string The parameter string is passed as 0-terminated string and contains the parameters according to the conventions of the action called.

Exist - Is instance selected?

The function checks for references and collections whether an instance is selected or not. For attributes the functions checks whether an instance area has been allocated for the instance or not.

```
logical PropertyHandle :: Exist ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

ExtractKey - Extract ident key value

The function extracts the ident key value according to the structure definition of the property handle.

When no instance has been passed the function extracts the key from the currently selected instance. If no instance has been selected in the property handle the function will set the cursor to the first instance (if possible). When no instance could be selected (empty datasource) the function returns an empty key.

When a key area has been passed in the key parameter the function returns the key in the passed key area. When no key or an empty key has been passed the returned key area refers to an internal area, which should not be modified by the application. This area is valid until the next ODABA interface function has been called.

```
Key PropertyHandle :: ExtractKey (Key ident_key_w, Instance in-
                    stance_w )
```

Return value	The key is provided in the internal key format. When necessary the key value can be converted to a string using the ({.r pib.KeyToString}()) function.
ident_key_w	Ident key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey() function. Default: Key() (empty key)
instance_w	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a reference to a propertly structured area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not. Default: Instance()

ExtractSortKey - Extract sort key value

The function extracts the key value according to the curent sort order (index).

When no instance has been passed the function extracts the key from the currently selected instance. If no instance has been selected in the property handle the function will set the cursor to the first instance (if possible). When no instance could be selected (empty datasource) the function returns an empty key.

When a key area has been passed in the key parameter the function returns the key in the passed key area. When no key or an empty key has been passed the returned key area refers to an internal area, which should not be modified by the application. This area is valid until the next ODABA interface function has been called.

```
Key PropertyHandle :: ExtractSortKey (Key sort_key_w, Instance
                  instance_w )
```

Return value	The key is provided in the internal key format. When necessary the key value can be converted to a string using the ({.r pib.KeyToString}()) function.
sort_key_w	Sort key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey() function. Default: Key() (empty key)
instance_w	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a reference to a propertly structured area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not. Default: Instance()

Fill - Fill instance from external one

The function copies all fields including references from the instance passed to the function into the selected instance.

```
logical PropertyHandle :: Fill (char *instance )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
instance	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

FillData - Fill instance from external one

The function copies all fields without references from the instance passed to the function into the selected instance.

```
logical PropertyHandle :: FillData (char *instance )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
instance	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

FirstKey - Locate first key

The function locates the first sort key in the index. When the data source is unordered the function locates the first instance and extracts the ident key.

The returned key area refers to an internal area, which should not be modified by the application. This area is valid until the next ODABA interface function has been called.

```
Key PropertyHandle :: FirstKey ( )
```

Return value

The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey}() function. When no key is passed by the application an empty key (without data area) will be passed.

Get - Get property instance

The Get() function allows selecting an instance in a property handle collection by key or position. The function can also be used to locate an instance in an array attribute or to get a single attribute instance. Before a new instance is selected the curent selection in the property handle is cancelled. in case of instance modifications on the previously selected instance those are saved automatically. Selecting an instance will also cancel the selection of all subordinated property handles.

The function returns the instance that has been selected in the property handle when is has been executed successfully. Otherwise it returns an empty instance.

i01 - Get Instance by index position

Reading an instance by position is locating the instance on the given location in the selected index (sort order). Thus, the result will change usually when changing the selected index for the collection. Using index positions for reading is also a weak point when indexes are updated simultaneously, since the index position might change when other users insert or remove entries from the index. To avoid this you may use the LockSet() function, that locks the complete collection. Access by position, however, is a comfortable way browsing through a collection. Access by position cannot be used for LOID or GUID property handles. Passing a number to an LOID handle interpretes the number as local object identity.

For a path property Get() by position automatically changes the selection for higher properties in the path when the end of collection is signaled an a level that is not the top level for the path property.

Passing AUTO as position the function returns the instance currently selected in the property handle. When no instance is selected the function returns the first instance in the collection. When an instance is selected in the property handle and the current access mode does not correspond to the required access mode, the function will re-read the instance. Thus, the function can be used to update the access mode when the selected instance was write protected. Selections in subordinated property handles are canceled when re-reading the instance.

```
Instance PropertyHandle :: Get (int32 set_pos0_w )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
set_pos0_w	Position in collection

The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance.

For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored.

Special positions are

CUR_INSTANCE

CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance.

FIRST_INSTANCE

FIRST_INSTANCE (0) refers to the first instance in a collection according to the selected index (sort order).

LAST_INSTANCE

FIRST_INSTANCE (0) refers to the last instance in a collection according to the selected index (sort order).

i02 - Get instance by key value

When reading an instance by key, this is considered as key in the currently selected index (SetOrder()). When the key cannor be found in the index the function returns an empty instance. It is also possible to locate an instance by key in an unordered collection, when it has been marked as unique (no duplicate instances). In this case the key must be passed according to the structure of the ident key. The key must be passed according to the internal key structure.

The key can also be an LOID or a GUID string when the property handle has been opened for reading by local object identities (__LOID) or by global unique identifiers (__GUID).

When positioning the instance for a path property the key must consist of all sort keys along the path.

```
Instance PropertyHandle :: Get (Key sort_key )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
sort_key	Sort key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey}() function. When no key is passed by the application an empty key (without data area) will be passed.

i03 - Get instance by value in the property handle

When passing the value for locating an instance in a collection via property hanndle, Get() is called as get by key or index depending on the data type defined in the property handle.

When the value property handle passes a structured instance of the same or a specialized type as the instance to be selected, the key value is extracted from the instance passed in the property handle and get by key is used.

When passing a text value (STRING, MEMO or CHAR), the text data is considered as string key where key komponents are separated by |. You cannot pass a structured key (Key) my means of property handles.

If not the key or instant version is used the 'get by position' version is called. Non-integer numerical values are rounded to the next lower integer number.

```
Instance PropertyHandle :: Get (PropertyHandle &prop_hdl )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

GetActionResult - Get result from last action executed

The function returns the resultstring from the last action executed. The result string is available until the next action call, only. When the action does not return a result the function returns NULL.

```
char *PropertyHandle :: GetActionResult ( )
```

Return value	The result string can be a list of strings where strings are usually separated by x01 characters. If there is only one string returned the string is 0-terminated. Multiple strings are terminated with 0 after the last string in the list, which should be terminated with x01 as well.

GetArea - Get Instance area

The function returns the instance area for the property handle. In contrast to GetInstance() the function returns the instance area as (char *) pointer regardless on whether an instance is selected in the property handle or not.

The function will always return a data area when the propertyhandle has a valid description. Hence it cannot be used for checking whether a data area is available. For checking whether a data area has been allocated use HasData().

i0 - Provide area for current property handle

The function returns the area for the current property handle.

```
char *PropertyHandle :: GetArea (char chkopt ) const
```

Return value	The instance area is structured according to the structure defiition (DBStructDef).
chkopt	Check option The option forces the function to check the property handle befor running executing the function. You can pass NO to avoid unnecessary checking.

i01 - Provide area for subsequent property

The function tries to locate the passed property and returns the area for this propertyinstance when the property exist. The function returns an error, when NULL or an empty property path is passed.

```
char *PropertyHandle :: GetArea (char *prop_path, char chkopt )
                         const
```

Return value	The instance area is structured according to the structure defiition (DBStructDef).
prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.
chkopt	Check option The option forces the function to check the property handle befor running executing the function. You can pass NO to avoid unnecessary checking.

GetAttribute - Get attribute according to position

The function returns the name of the attribute according to the given index. The attribute index corresponds to the definition of the structure. Attributes returned are always attributes with basic (elementary) types.

Attributes in base structures or imbedded structures are returned as property pathes (e.g. address.city when address is a structured attribute in person). The leading part for base structures (usually the structure name) is displayed only when passing YES for the full_path option (default: NO).

Generic attributes can be considered as references or as attributes. The generic option defines whether generic attributes are considered as attibutes (default: YES).

If there are no attributes defined for the structure the function returns NULL. When an attribute with the given index has been found the function returns the property path in the fldpath.

The function returns the path in communication area of the property handle. This area is destroyed when calling the next PropertyHandle function.

```
char *PropertyHandle :: GetAttribute (int32 indx0, logical
                    full_path, logical generic )
```

Return value	The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.
indx0	Position in collection The position in the collection addresses the first instance in the collection with 0. AUTO (-1) refers to an undefined position.
full_path	Full path The full path option is used to request the property path including base structure names. Default: YES
generic	Generic attruibute option The option allows considering generic attributes as references. Default: YES

GetBaseProperty - Get collection handle for base collection

The function returns the base collection property handle when a base collection has been defined for the collection (relationship or extent). Otherwise the function returns NULL.

The returned property handle is a sub-handle for the current property handle, i.e. the base collection will change automatically whenever the collection in the current property handle changes.

```
PropertyHandle *PropertyHandle :: GetBaseProperty ( )
```

Return value	Is a pointer to an (usually) opened property handle.

GetBufferInstance - Read instance from Buffer

The function reads an instance from the property handle buffer. The function returns the instance according to the buffer position indx0. The first instance in the buffer is addressed by 0. Usually the buffer index is not identical with the position in the collection.

```
Instance PropertyHandle :: GetBufferInstance (int32 indx0 )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
indx0	Position in collection The position in the collection addresses the first instance in the collection with 0. AUTO (-1) refers to an undefined position.

GetCollectionID - Returns local collection identity

Collections in ODABA have a unique identity like instances, which allows identifying a collection within a database.

```
int32 PropertyHandle :: GetCollectionID ( )
```

Return value

GetCollectionProperty - Get parent collection property handle

The function searches for the next higher collection property handle. When the property handle is a top handle or a transient property handle the function returns NULL.

```
PropertyHandle *PropertyHandle :: GetCollectionProperty ( )
```

Return value	Is a pointer to an (usually) opened property handle.

GetCount - Get number of instances stored for property

The functio returns the number of instances stored in the collection. Since some indexes store multiple references to instances (array index) or do not store all instances in the index, the function returns rather the number of references in the index than the number of instances in the collection. Usually these numbers are, however, identical. To ensure that you get the collection cout you may select the default order (SetOrder()).

```
int32 PropertyHandle :: GetCount ( )
```

Return value

GetCurrentIndex - Get cursor position

The function returns the position of the currently selected instance in the collection according to the selected sort order. The function returns a value when an instance has been located in the property handle (e.g. LocateKey()) regardles whether the instance has been red or not.

Since the position might change when instances are inserted or removed from the collection the current index can be used for accessing instances (Get(indx0)) in a limited way (e.g. within a transaction).

```
int32 PropertyHandle :: GetCurrentIndex ( )
```

Return value — The position refers to the position of an instance reference in a local or global collection. The instance position refers to the position according to the selected index (sort order, -> {.r pib.SetOrder}()).

If the selected index is not unique the system decides the order among instances with the same key value.

The position may change when inserting or deleting instances (e.g. in other applications)..

GetCurrentSize - Get size for selected instance

The function usually returns the same as GetSize. For weak typed collections, however, the function returns the size of the selected instance in staed of the defined type for the collection.

```
uint32 PropertyHandle :: GetCurrentSize ( )
```

Return value — Size of the instance or property area.

GetCurrentType - Get type for selected instance

The function usually returns the same as GetType. For weak typed collections, however, the function returns the type of the selected instance in staed of the defined type for the collection.

```
char *PropertyHandle :: GetCurrentType ( )
```

Return value — The type name is passed as 0-terminated string with a maximum length of 40 significant characters.

GetCurrentTypeDef - Get current type definition

The function usually returns the same as GetType. For weak typed collections, however, the function returns the type definition of the selected instance in staed of the defined type for the collection.

```
DBStructDef *PropertyHandle :: GetCurrentTypeDef ( )
```

Return value	The structure definition is provided in the internal format as pointer to a DBStructDef object.

GetDBHandle - Get database handle

The function returns the database handle for the current property handle.

```
DatabaseHandle &PropertyHandle :: GetDBHandle ( )
```

Return value	This is pointer to an opened database handle. The database handle can be an opened database handle DatabaseHandle as well as an opened dictionary handle (DictionaryHandle).

GetDate - Get Date value for property handle

The function returns the date value for the current property handle or for the attribute passed in prop_path.

i0

```
dbdt PropertyHandle :: GetDate ( )
```

Return value	The data value is passed in the internal data format.

i01

```
dbdt PropertyHandle :: GetDate (char *prop_path )
```

Return value	The data value is passed in the internal data format.
prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

GetDateTime - Get property instance as time stamp (date/time)

The function returns the date/time value for the current property handle or for the attribute passed in prop_path.

i0

```
dttm PropertyHandle :: GetDateTime ( )
```

Return value	A date-time value or time point is passed in the internal date-time format.

i01

```
dttm PropertyHandle :: GetDateTime (char *prop_path )
```

Return value	A date-time value or time point is passed in the internal date-time format.
prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

GetDescription - Get property definition

The function returns the field definition for the forperty handle. The field definition described the structure of the instance area for the property handle.

If the property handle is a collection the field definition describes one instance of the collection according to the defined structure. This is different from the field definition of the property handle itself.

```
DBFieldDef *PropertyHandle :: GetDescription (char chkopt )
                    const
```

Return value	The property defintion contains the metadata for the referenced property instance.
chkopt	Check option The option forces the function to check the property handle befor running executing the function. You can pass NO to avoid unnecessary checking.

GetDictionary - Get dictionary handle

The function returns the dictionary handle for the current property handle.

```
DictionaryHandle &PropertyHandle :: GetDictionary ( )
```

Return value	An opened dictionary handle is passed.

GetDimension - Provide field dimension

The function returns the property dimension.

In case of an error the function returns -1 (AUTO).

```
int32 PropertyHandle :: GetDimension ( ) const
```

Return value	The dimension describes the property dimension. this is the maximum number of instances that can be stored for the property. The function returns 0 (UNDEF) if there is no limit (collection) or the dimension (cardinality) defined for the property.

GetDouble - Get property instance as double value

The function returns the value for the current property handle or for the attribute passed in prop_path as double value .

i0

```
double PropertyHandle :: GetDouble ( )
```

Return value

i01

```
double PropertyHandle :: GetDouble (char *prop_path )
```

Return value	
prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

GetExtentName - Get extent name for collection

The function returns the name or path for the base collection (extent name or property path). When no ase collectin has been defined the function returns NULL.

```
char *PropertyHandle :: GetExtentName ( )
```

Return value	The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

GetFieldDef - Get field definition for the property

The function returns the field definition for the property handle (which differs in case of collection properties from the instance definition, which is returned by GetDescription()).

```
DBFieldDef *PropertyHandle :: GetFieldDef ( )
```

Return value	The property defintion contains the metadata for the referenced property instance.

GetGUID - Get global identity string for the current instance

The function returns the global instance identity (GUID) for the current instance. This identity is unique within all ODABA2 databases. GUIDs are available for instances that are derived directly or indirectly from __OBJECT. When auto-build (GUID) is set for the structure the GUID is generated when creating the instance. Otherwise it has to be provided using ProvideGUID(). When no GUID has been defined for the structure (not derived from __OBJECT) the function returns the local object identity (LOID), which is a unique identifier within the database.

If no instance is available or no global identity has been generated for the current instance the function returns NULL.

The function returns a global instance identity also when the property handle refers to a new instance where the global identity has been set explicitely. Thus, e.g. when copying instances you might ask for the global identity that has been shipped with the source instance.

The GUID is passed in the internal result area and valid until the next property handle function call.

```
char *PropertyHandle :: GetGUID ( )
```

Return value	The global instance identifier is passed as 0-terminated string with a maximum length of 40 characters.

GetGenAttrType - Get generic attribute type

The functio returns the internal type value of the generic attribute, that has been selected as current attribute type (SetGenAttribute()).

```
int32 PropertyHandle :: GetGenAttrType (char *w_propnames )
```

Return value	The type for a generic attribute is a valid value from the basic enumeration of the generic attribute. UNDEF (0) indicates an undefined generic type.
w_propnames	Property path or name The property name is passed as 0-terminated string. It may contain a property path that consists of a sequence of property names. NULL is passed if no property name is available.

GetGenOrderType - Get current type for generic sort order of collection handle

When the key of the currently selected sort order contains a generic attribute the index is generic as well. In this case setting the sort order implies setting it to a specific type of the generic attribute. This function returns the internal type number for the generic attribute type selected for the given sort order (SetOrder()).

```
int32 PropertyHandle :: GetGenOrderType ( )
```

Return value	The type for a generic attribute is a valid value from the basic enumeration of the generic attribute. UNDEF (0) indicates an undefined generic type.

GetGlobalID - Get global ID

The function returns the local object ID (LOID) when the instance could be found in the base collection (global extent). When no key is passed the instance is searched with the key from the internal instance. When passing a key and an instance is selected in the property handle the instance wil be unselected.

i00

```
int32 PropertyHandle :: GetGlobalID ( )
```

Return value	The local object identity is a 31-bit number that identifies an object instance uniquely in a database. LOIDs are available only for independently stored instances but not for instances of imbedded structures.

i01

```
int32 PropertyHandle :: GetGlobalID (void *skey )
```

Return value	The local object identity is a 31-bit number that identifies an object instance uniquely in a database. LOIDs are available only for independently stored instances but not for instances of imbedded structures.
skey	Sort key The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the ({.r pib.StringToKey}()) function. Regardles on the type key values are passed as (char *) areas.

GetIdentity - Get property identity string

The function builds an identity string for a property in a structure instance:

loid|property_path[index]

The id-string is a local database identity for the instance. 'property_path' is the path that identifies a property within the instance. When the property is an array the element position is indicated by the index. Index numbers may also appear within the path (e.g. when supporting upto three addresses for a person a property path for 'place" could look like:

199879899|address[2].place

The identity string is returned in an internal area when no area is passed (id_string). Otherwise the area passed in string should have 513 bytes, but at least the maximum expected string size +1.

If there is no instance selected in the (upper) collection property handle the function returns NULL. If no id_string is passed the function returns the path in communication area of the property handle. This area is destroyed when calling the next PropertyHandle function.

```
char *PropertyHandle :: GetIdentity (char *id_string )
```

Return value	String that identifies a property uniquely in the database.
id_string	Identity string String that identifies a property uniquely in the database.

GetIndexName - Name of the current index

The function returns the index name of the index for the n-th (indx0) index (sort order for collection). The function returns NULL, if indx0 is equal or larger than the number of indexes defined for the collection.

```
char *PropertyHandle :: GetIndexName (int32 indx0 )
```

Return value	The index or key name refers to an index defining an order for the collection. Indexes are always referenced by key names.
indx0	Position in collection

The position in the collection addresses the first instance in the collection with 0. AUTO (-1) refers to an undefined position.

GetInitArea - Provide initialized instance area

The function provides an initialised instance of the type defined for the property handle. For weak typed collection the type depends on the last type accessed by the property handle or the type set with the SetType() function. Calling the function for an attribute will return an initialized attribute instance.

When the property handle has selected an instance the current selection is cancelled (after saving changes made to the selected instance). You may fill attributes and initialise single references but you cannot add instances to collections in an initialised instance.

For storing the instance to the database you must call the Add() function for the property handle. Changes made to the instance are not saved automatically when changing the selection for the property handle. Since no instance is selected in the property handle after GetInitArea() you cannot assign values to subordinated property handles.

```
Instance PropertyHandle :: GetInitArea ( )
```

Return value

Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area.

You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

GetInitInstance - Provide initialised instance

The function provides an initialised instance of the type defined for the property handle. For weak typed collection the type depends on the last type accessed by the property handle or the type set with the SetType() function. Calling the function for an attribute will return an initialized attribute instance.

When the property handle has selected an instance the current selection is cancelled (after saving changes made to the selected instance). You may fill attributes and initialise single references but you cannot add instances to collections in an initialised instance.

The instance can be stored to the database by calling the Save() function for the property handle. Changes made to the instance are saved automatically when changing the selection for the property handle. To avoid storing the instance to the database it must be explicitly cancelled.

```
Instance PropertyHandle :: GetInitInstance ( )
```

Return value — Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area.

You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

GetInstModCount - Get update count for selected instance

The function returns an internal modification count for an instance. This allows checking whether the instance has been updated by another user or application, since each update will increase the modification count stored in the database.

Since the modification count is rotating (starting again with 1 after reaching 255) this is not a save indication. Thus, it is suggested to use server event handler for reacting on changes.

```
int16 PropertyHandle :: GetInstModCount ( )
```

Return value — The modification count contains the number of modifications for an instance. After 255 modifications it starts to count from the beginning. Only modifications that are written to database are counted.

GetInstance - Get current instance

The function returns the current instance. If no instance is selected the function returns an empty instance area (instance.GetData() returns NULL).

```
Instance PropertyHandle :: GetInstance ( ) const
```

Return value

Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area.

You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

GetInstanceContext - Get Instance Context

The function returns the Instance context for the client or server depending on where the property handle is being created.

```
CTX_Structure *PropertyHandle :: GetInstanceContext ( )
```

Return value

This is the default structure context or a user-defined context class instance for the structure.

GetInt - Get property instance as integer value

The function returns the integer value for the current property handle or for the attribute passed in prop_path. When the field definition refers to the value with decimal precisions the value returned contains only the part before the decimal point. To get the exact value use GetNormalized().

i0 - Get ineger value for current property

This implementation provides an integer value for the selected property handle.

```
int32 PropertyHandle :: GetInt ( )
```

Return value

i01 - Get integer value for passed attribute

This implementation returns the integer value for the property addressed in the property path (prop_path). The attribute passed must be a valid attribute in the structure of the current property handle.

You may also pass "__IDENTITY" or "__LOID" to obtain the local identity value (database identity) for the instance.

```
int32 PropertyHandle :: GetInt (char *prop_path )
```

Return value

prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

GetIntValue - Get property instance as integer value

The function returns the integer value for the current property handle or for the attribute passed in prop_path. When the field definition refers to the value with decimal precisions the value returned contains only the part before the decimal point. To get the exact value use GetNormalized().

In contrast to GetInt(), no instance must be selected in the property handle.

i0

```
int32 PropertyHandle :: GetIntValue ( )
```

Return value

i01

```
int32 PropertyHandle :: GetIntValue (char *prop_path )
```

Return value

prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

GetKeyLength - Get ident key length

The function returns the identifying key length.

```
int16 PropertyHandle :: GetKeyLength ( )
```

Return value	This is the size for the internal (structured) key according to the attributes composing the key.

GetKeySMCB - Get ident key definition

The function returns the structure defintion for a key defintion of the structure defined for the property handle. This is not necessaryly an index or sort key of the collection referenced by the property handle. When no key is passed the function returns the key definition of the identifying key.

```
smcb *PropertyHandle :: GetKeySMCB (char *key_name_w )
```

Return value	The smcb is a more general way to define structure (DBStructDef). It contains information for the structure and its properties. In contrast to the DBStructDef the smcb describes structure members regardless on the rule they may play in the structure.
key_name_w	Key name for conversion The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks. If no key name is passed (NULL) the sort key according to the selected sort order is used instead,

GetLOID - Get instance identity (LOID)

The function returns the local object identity for the instance selected in the property handle. When no instance is selected the function returns 0.

When passing a key or a position the instance acording to the passed key or position is selected before retrieving the LOID.

i0

```
int32 PropertyHandle :: GetLOID (int32 set_pos0_w )
```

Return value	The local object identity is a 31-bit number that identifies an object instance uniquely in a database. LOIDs are available only for independently stored instances but not for instances of imbedded structures.
set_pos0_w	Position in collection The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance. For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored. Special positions are **CUR_INSTANCE** CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance. **FIRST_INSTANCE** FIRST_INSTANCE (0) refers to the first instance in a collection according to the selected index (sort order). **LAST_INSTANCE** FIRST_INSTANCE (0) refers to the last instance in a collection according to the selected index (sort order).

i1

```
int32 PropertyHandle :: GetLOID (Key sort_key )
```

Return value	The local object identity is a 31-bit number that identifies an object instance uniquely in a database. LOIDs are available only for independently stored instances but not for instances of imbedded structures.
sort_key	Sort key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey}() function. When no key is passed by the application an empty key (without data area) will be passed.

GetMode - Get access mode for collection handle

The function returns the access mode for the property handle as defined when creating or opening the property handle or as set with the ChangeMode() function.

```
PIACC PropertyHandle :: GetMode ( )
```

Return value	Access mode that has been set for the property handle.

GetNormalized - Get normalized integer value

The function can be used for getting integer values with decimal precisions from INT or unsigned INT attributes. When defining a an attribute with two decimals, referring to the value 1, which is stored inernally as 100 (1.00), will result in 1. Using GetNormalized will result in 100 instead.

```
int32 PropertyHandle :: GetNormalized ( )
```

Return value	The value is passed as platform independent 32-bit integer value.

GetObjectHandle - Get Database Object handle

The function returns the Database Object Handle for the property handle. When referring to transient fields, which are not associated with a database object, the function returns NULL.

For transient fields the function returns the database object handle that is associated with the parent instance and not the database object handle of the associated collection, which might in some cases belong to another database object.

```
DBObjectHandle &PropertyHandle :: GetObjectHandle ( )
```

Return value	The database object handle defines a database object (subject) within the database. Each database has at least a root object, which might be identically with the database.

GetOrigin - Get associated property handle

The function returns an ID that identifies the origin of a property handle. For transient property handles thie is the origin for the associated property handle. For copy handles it is the origin of the copy.

The function returns UNDEF (0) when the handle is not opened or when no property handle is associated with a transient property handle.

```
int PropertyHandle :: GetOrigin ( )
```

Return value	The handle identifier allows comparing whether property handles refer to the collection or property.

GetParentProperty - Get high property

The function returns the next higher property handle. The handle returnd is shared with other handles in the application, which asked for the parent property handle. To get a private copy you can use the copy constructor:

ph(GetParentProperty());

GetParentProperty returns a so-called static property handle, which is not able to react on type changes in upper weak-typed property handles. To avoid problems with static property handles (error 348) create a shared property handle as described above.

```
PropertyHandle *PropertyHandle :: GetParentProperty ( )
```

Return value	Is a pointer to an (usually) opened property handle.

GetPrivilege - Get access privilege for reference

The function returns the access privilege for the proper-ty.

```
PIADEF PropertyHandle :: GetPrivilege ( )
```

Return value

The access privilege describes the accessability of the property.

ODC_privat - accessable within the class, only

ODC_protected - accessable from outside via get_-functions

ODC_public - accessable from outside without re-strictions

ODC_undefined - accessability not defined

GetPropertyContext - Get property context

The function returns the property context for the client or server depending on where the property handle is being created.

```
CTX_Property *PropertyHandle :: GetPropertyContext ( )
```

Return value

This is the default property context or a user-defined context class instance for the property.

GetPropertyHandle - Get property handle

The function returns the property handle for the selected path. The function handles property pathes within a structure instance (e.g. 'direction.city', where city is a member of the imbedded 'Address' structure of direction) as well as pathes that include references ('mother.name', where mother is a reference to a persons mother). When defining pathes that include references thous references shout be single referenced (dimension = 1) since the path will locate the first instance for the parent(s), only.

When referring to transient references you must take into accont thet the data source referenced by the transient reference may change during processing. This includes the type of referenced instances as well as the referenced collection or instance.

When using the GetPropertyHandle function instead of using a property handle constructor you will share the cursor and the data source with other property handles provided with this function.

For creating a shared subordinated property handle you can use the constructor in combination with the GetPropertyHandle() function:

PropertyHandle ph(parent.GetPropertyHandle(prop_path));

or the corresponding Open() function.

GetPropertyHandle returns a so-called static property handle, which is not able to react on type changes in upper weak-typed property handles. To avoid problems with static property handles (error 348) create a shared property handle as described above.

i00

```
PropertyHandle *PropertyHandle :: GetPropertyHandle (char
                 *prop_path, logical *is_transient )
```

Return value	Is a pointer to an (usually) opened property handle.
prop_path	Property path

The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

i01

```
PropertyHandle *PropertyHandle :: GetPropertyHandle (char
                   *prop_path )
```

Return value	Is a pointer to an (usually) opened property handle.
prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

GetPropertyPath - Get property path for property handle

The property path defines the path from the parent property handle to the current one.

```
char *PropertyHandle :: GetPropertyPath ( )
```

Return value	The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

GetRefModCount - Get collection update count

The function returns an internal modification count for a collection. This allows checking whether the collection has been updated by another user or application by adding or deleting instances from teh collection, since each update will increase the modification count stored in the database.

Since the modification count is rotating (starting again with 1 after reaching 255) this is not a save indication. Thus, it is suggested to use server event handler for reacting on changes.

```
int16 PropertyHandle :: GetRefModCount ( )
```

Return value	The modification count contains the number of modifications for an instance. After 255 modifications it starts to count from the beginning. Only modifications that are written to database are counted.

GetReference - Get reference from structure definition

The function returns the name of the reference or relationship (except MEMO fields that are considered as attributes) according to the given index. The reference index corresponds to the definition of the structure.

References in base structures or imbedded structures are returned as property pathes (e.g. address.city when address is a structured attribute in person and city is a reference in address). The leading part for base structures (usually the structure name) is displayed only when passing YES for the full_path option.

Generic attributes can be considered as references or as attributes. The generic option defines whether generic attributes are considered as references.

If there are noreferences defined for the structure the function returns NULL. When an reference with the given index has been found the function returns the property path in the fldpath.

The function returns the path in communication area of the property handle. This area is destroyed when calling the next PropertyHandle function.

```
char *PropertyHandle :: GetReference (int32 indx0, logical
                        full_path, logical generic )
```

Return value	The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.
indx0	Position in collection The position in the collection addresses the first instance in the collection with 0. AUTO (-1) refers to an undefined position.
full_path	Full path The full path option is used to request the property path including base structure names. Default: YES
generic	Generic attruibute option The option allows considering generic attributes as references. Default: YES

GetSelectedKey - Get selected key value

The function returns the key value for the selected instance. When no instance is selected the function returns the value for the selected key, which might have been located wit the LocateKey() or NextKey() function.

When neither a key nor an instance is selected, the function returns an empty key instance.

```
Key PropertyHandle :: GetSelectedKey ( )
```

Return value	The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey}() function. When no key is passed by the application an empty key (without data area) will be passed.

GetSize - Get instance size

The function returns the size allocated for the instance.

```
int32 PropertyHandle :: GetSize ( )
```

Return value

GetSizeOf - Get size of instance in collection handle

The function returns the size allocated for the instances of the property handle (for weak typed the size for the selected type). For MEMO fields the function returns the maximum size of the MEMO field.

```
int32 PropertyHandle :: GetSizeOf ( )
```

Return value

GetSortKeyLength - Get sort key length

The function returns the length of the selected sort key. When the collection is unordered the function returns 0.

```
int16 PropertyHandle :: GetSortKeyLength ( )
```

Return value	This is the size for the internal (structured) key according to the attributes composing the key.

GetSortKeySMCB - Get sort key definition

The function returns the key definition for the selected sort key.

```
smcb *PropertyHandle :: GetSortKeySMCB ( )
```

Return value	The smcb is a more general way to define structure (DBStructDef). It contains information for the structure and its properties. In contrast to the DBStructDef the smcb describes structure members regardless on the rule they may play in the structure.

GetString - Get property instance as string value

The function returns the string value for the current property handle or for the attribute passed in prop_path. In addition to explicite properties defined for the structure of the instance '__LOID' and '__GUID' can be passed as property path (prop_path) for retrieving the local object identifier and the global unique identifier for the instance.

The function returns always a pointer to a valid string. When the requested attribute is not available the string length is 0.

i0 - Get string for current property

The function returns the string value for the current property.

```
char *PropertyHandle :: GetString ( )
```

Return value	Pointer to the 0-terminated string area.

i01 - Get String for referenced property

The function tries to locate the passed property (prop_path) and returns the string value when the property exist. The function returns an error, when NULL or an empty string is passed as property name.

You may also pass "__IDENTITY" or "__LOID" to obtain the local identity value (database identity) for the instance. Passing "__GUID" will return the global unique identifier, when being defined for the object instance.

```
char *PropertyHandle :: GetString (char *prop_path )
```

Return value	Pointer to the 0-terminated string area.
prop_path	Property path

The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

GetStringLength - Provide stringlength for instance

The function returns the maximum lengts of the string that results from converting the instance into a string.

```
uint32 PropertyHandle :: GetStringLength ( )
```

Return value	Size of the instance or property area.

GetStringValue - Get property instance as string value

The function returns the string value for the current property handle or for the attribute passed in prop_path. In addition to explicite properties defined for the structure of the instance '__LOID' and '__GUID' can be passed as property path (prop_path) for retrieving the local object identifier and the global unique identifier for the instance.

In contrast to GetString() the no instance must be selected in the property handle.

i0

```
char *PropertyHandle :: GetStringValue ( )
```

Return value	Pointer to the 0-terminated string area.

i01

```
char *PropertyHandle :: GetStringValue (char *prop_path )
```

Return value	Pointer to the 0-terminated string area.
prop_path	Property path

The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

GetStructDef - Get structure definition

The function returns the structure definition for the instances in the collection. For weak typed collection the function returns the base structure definition defined for the weak typed collection.

```
DBStructDef *PropertyHandle :: GetStructDef ( ) const
```

Return value	The structure definition (DBStructDef) contains the metadata for the instance, i.e. information for the structure and its properties.

GetTime - Get property instance as time value

The function returns the time value for the current property handle or for the attribute passed in prop_path.

i0

```
dbtm PropertyHandle :: GetTime ( )
```

Return value	The time value is passed in the internal data format.

i01

```
dbtm PropertyHandle :: GetTime (char *prop_path )
```

Return value	The time value is passed in the internal data format.
prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

GetType - Get basic collection type

The function returns the type name for the instances in the collection. For weak typed collection the function returns the name of the base structure defined for the weak typed collection.

```
char *PropertyHandle :: GetType ( )
```

Return value	The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.

GetValue - Get instance value

The function returns the value for the property instance. It should be used for accessing attribute handles. In contrast to Get the function returns an instance area also when the attribute handle is not positioned (e.g. for the initial instance before creating an instance).

For collection properties the function works the same way as the Get() function except when passing AUTO as index value. In this case GetValue returns the instance area also when no instance is selected.

When accessing array attributes the function returns the array element according to the passed index.

```
Instance PropertyHandle :: GetValue (int32 lindx0 )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
lindx0	Position in collection

GetVersion - Get version number for selected instance

The function returns the version number for the instance selected. Since instance versions are created only in case of updates the requested version in the property handle might be higher than the version returned from the current instance. The function returns the version number, only. Fo determining the time period you can call the ACObject::VersionIntervall() function when using database versions.

```
uint16 PropertyHandle :: GetVersion ( )
```

Return value	Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION

Group - Grouping operation

The grouping operation allows grouping a collection of instances according to a condition, key or list of attributes. The instances grouped are collected in a property named 'partition', which has the same type as the input collection for the operation.

When defining conditional values in the grouping rule, a string attribute named 'value' is created for each output instance. Otherwise, the output contains the attributes defined in the attribute list or composing the key. When passing an ODABA OQL expression as grouping rule, the type of the attribute in the target depends on the type returned by the expression.

When defining conditional values, the instance is associated with the first value, that matches the condition when passing distinct YES. Otherwise, the instance is associated with each value that matches the condition, which might create duplicates.

When the calling property handle refers to a non empty collection all instances are removed before performing the operation. When the calling property handle is empty the function creates a temporary extend for storing the result. You may change the buffer size for the target property handle to increase the performance of the operation. This is not necessary, when you group by sort key.

```
PropertyHandle &PropertyHandle :: Group (PropertyHandle
                    &prophdl_ref, char *grouping_rule, logical
                    distinct )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.
grouping_rule	Grouping rule

The grouping rule describes the grouping strategy. You may either group instances accrding to the values for a list of attributes or a key, which is considered as attribute list as well (country, city). You may also define attribute values by means of conditions (low: income < 1000, medium: income < 3000, high). You may also pass an ODABA OQL expression, which will get tht name 'value', too.

distinct	Distinct option Passing a distinct option YES forces the function to remove duplicates from the result collection.

HasData - Is data available for property

The function returns true (YES) for an collection property handle when an instance is selected. For all other property handles the function returns true when a parent property handle exists with a selected instance or when no parent property handle exist (independent property handle) and a data area has been assigned to the property handle.

```
logical PropertyHandle :: HasData ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

HasDescription - Is description available

The function returns true (YES) when a description exists for the property handle, NO otherwise.

```
logical PropertyHandle :: HasDescription ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

HasGenericAttributes - Does the instance have generic attributes

The function returns true (YES) when the instance has generic attributes and false (NO) otherwise.

```
logical PropertyHandle :: HasGenericAttributes ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

HasIndex - Does a collection have an index?

The fucntion checks whether an index is defined for the property handle or not. When an index is defined this does not necessarily mean that the collection is ordered, since there are also indexes for unordered collections.

```
logical PropertyHandle :: HasIndex ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Initialize - Initialise instance area

The function initializes the area of the passed instance with the defined default values. When no instance is passed the internal instance area for the property handle instance is initialized.

This function does not work for simple property handles (e.g. string property handle as PH("string")). For initializing simple property handle you must explicitly pass the instance area (ph.Initialize(ph.GetArea()).

```
logical PropertyHandle :: Initialize (Instance instance_w )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
instance_w	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a reference to a propertly structured area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not. Default: Instance()

InsertTerminator - Insert line terminator for large text fields

The function inserts the terminator string (string) at the end of the text field.

```
logical PropertyHandle :: InsertTerminator (char *string )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
string	String area Pointer to the 0-terminated string area.

Intersect - Intersect collections

The result collection contains all instances that are contained in each operand collections. The existence of an instance in a collection can be checked based on the sort key (passing YES for ik_opt) or on local identities (LOID). Using the LOID is save but comparing the key is much faster. Hence, the key check should be used whenever possible.

Calling the function with one operator creates the intersection between the calling and the passed collection and stores the result in the calling collection. Otherwise the operation is performed with the passed operands storing the result in the collection referenced by the calling property handle. When the calling property handle refers to a non empty collection all instances are removed before performing the operation. When the calling property handle is empty the function creates a temporary extend for storing the result.

i0

```
PropertyHandle &PropertyHandle :: Intersect (PropertyHandle
                     &prop_hdl1, PropertyHandle &prop_hdl2, char
                     sk_opt )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prop_hdl1	First Property handle Reference to an opened property handle.
prop_hdl2	Second Property handle

	Reference to an opened property handle.
sk_opt	Sort key option The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances. Default: YES

i1

```
PropertyHandle &PropertyHandle :: Intersect (PropertyHandle
                **ph_list, int16 count, char sk_opt )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
ph_list	List of property handles An array of property handles acting as operands in the operation. The number of property handles in the array is passed in the count-parameter.
count	Number of entries
sk_opt	Sort key option The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances. Default: YES

i2

```
PropertyHandle &PropertyHandle :: Intersect (PropertyHandle
                &prophdl_ref, char sk_opt )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.
sk_opt	Sort key option

The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances.

Default: YES

IsActive - Is property an active property

An active property is a property that is able to react on events. Active properties are treated in a special way since events are generated for several occations.

```
logical PropertyHandle :: IsActive ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsAttribute - Is property an attribute?

The function returns whether the property handle refers to an attribute (YES) or not (NO). The functio returns also NO for shared base structure instances, which are considered rather as relationships than attributes. Imbedded base structure instances are, however, considered as attributes. The function returns NO also for generic attributes, which are considered as references.

```
logical PropertyHandle :: IsAttribute ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsBasedOn - Is structure derived from passed type?

The functio returns whether the current structure of the property handle is a specialization of the structure passed in strnames (YES) or not (NO). When no instance is selected thr function evaluates the structure defined for the property handle.

When the current structure is identical with the structure passed in strnames the function returns NO.

```
logical PropertyHandle :: IsBasedOn (char *strnames ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
strnames	Structure name

The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.

IsBasetypeProperty - Is property member of the base type

The function checks for members of weak typed instances whether the property is part of the common base type (YES) or not (NO).

```
logical PropertyHandle :: IsBasetypeProperty ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsBasicType - Is the type of the PropertyHandle an elementary one ?

The Type of the Property Handle is either a basic type or an enumeration or a structure. The function returns YES when the property refers to a basic type and NO otherwise.

```
logical PropertyHandle :: IsBasicType ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsClient - Is property handle a client handle

The function returns whether the property handle has been created for a client or a local application. The function returns NO, when the handle has been created on the server side in a client server environment.

```
logical PropertyHandle :: IsClient ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsCollection - Is property a collection or reference?

The function returns whether the property handle refers to a collection (YES) or not (NO). References and relationships are always considered as collections, as well as generic attributes, extents and views.

```
logical PropertyHandle :: IsCollection (char chkopt ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
chkopt	Check option The option forces the function to check the property handle befor running executing the function. You can pass NO to avoid unnecessary checking.

IsCollectionUpdate - Can collection be updated

The function returns whether the collection can be updated, i.e. whether instances can be added, renamed or removed or deleted from the collection (YES) or not (NO).

```
logical PropertyHandle :: IsCollectionUpdate ( ) const
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

IsCopyHandle - Is property handle a copy handle

The function returns whether the property handle is a copy from another handle (YES) or not (NO). Copy handles are created based on a property handle but using an own cursor. Copy handles are closed automatically when its origin is closed.

```
logical PropertyHandle :: IsCopyHandle ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsEmpty - Is property instance empty?

The function returns whether the property handle is empty (YES) or not (NO). A property is considered as emty when it is a collection with no instances or (if it is not a collection) when:

- the value is false (LOGICAL)

- the value is 0 (INT, REAL, Enumeration, DATE, TIME, DATETIME)

- the value is 0, ' ' oder 'N' (CHAR, STRING,CCHAR,MEMO)

- when all values for a structured instance are empty

```
logical PropertyHandle :: IsEmpty ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsEnumeration - Is the type of the PropertyHandle an enumeration ?

The Type of the property handle is either a basic type or an enumeration or a structure.The function returns true when the property handle refers to an enumeration.

```
logical PropertyHandle :: IsEnumeration ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsInitInstance - In instance initialized instance

When the instance has been provided with the GetInitInstance() function the instance has not yet been created and access is limited for sub-ordinated property handles. The function returns whether the selected instance in the property handle is an initialised instance or not.

```
logical PropertyHandle :: IsInitInstance ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsMemo - Check property type for memo field

The function returns true (YES) when the property refers to a large text field (MEMO).

```
logical PropertyHandle :: IsMemo (char chkopt ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
chkopt	Check option The option forces the function to check the property handle befor running executing the function. You can pass NO to avoid unnecessary checking.

IsNewInstance - Is new instance

The function returns whether the instance has just been created (YES) or not (NO). The "new instance" state is changed when the new instance is stored the first time after creating it or when re-reading it.

Since an instance may consist of several base structure instances that are stored independently (shared base structures) the ney instance state for such base structures may differ from the instance state, when the instance is indicated as new instance.

```
logical PropertyHandle :: IsNewInstance ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsNumeric - Check property type for numeric

The functio returns true (YES) when the property handle describes a numeric value (INT, UINT, REAL).

```
logical PropertyHandle :: IsNumeric ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsPositioned - Is instance positioned

The function checks whether there is an instance selected for the property handle. The function returns YES if an existing instance is selected, NO otherwise. When the property handle refers to an attribute with a parent property handle it returns the state of the parent instance.

When an instance has been provided using the GetInitInstance() function the property handle is not positioned. For checking whether an existing or new instance is selected or not, the IsSelected() function can be used.

```
logical PropertyHandle :: IsPositioned (char chkopt ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
chkopt	Check option The option forces the function to check the property handle befor running executing the function. You can pass NO to avoid unnecessary checking.

IsReadOnly - Is property read only

There are several reasons for properties getting the read-only state. An attribute is read-only when the instance it belongs to has been set to read-only for some reason (used by another user, cannot be updated by current user or others). Collections are usually set to read-only when the instance the collection belongs to has been set to read only, but they might also be persistent write protected (e.g. when being locked in workspaces). Since MEMO fields are stored as separate instances, a MEMO field can be read-only even thought the parent instance can be updated (e.g. when being locked in a transaction). Moreover, any property can be set to read-only by means of the context function SetReadOnly() in the application.

Attributes and MEMO fields with read-only state cannot be updated. Collection properties that are read-only do not allow inserting, renaming or removing instances from the collection.

```
logical PropertyHandle :: IsReadOnly ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsSensitive - Is property handle sensitive against modifications

A sensitive property is a property that is part of keys used in global indexes. In ODABA modifications on indexes will lock the index until the transaction is terminated. Thus, updating sensitive properties in long transactions may cause uncomfortable lock situations. Sensitive properties should not be allowed being updated in long transactions.

```
logical PropertyHandle :: IsSensitive ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsServer - Is property handle a server handle

The function returns whether the property handle has been created for a server or a local application. The function returns NO, when the handle has been created on the client side in a client server application.

```
logical PropertyHandle :: IsServer ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsShareBaseHandle - Is property handle for base structure

The function returns whether the property handle is a handling an independent (shared) base structure (YES) or not (NO). The function returns NO for all other property handles, imbedded base structures and in case of error.

```
logical PropertyHandle :: IsShareBaseHandle (char chkopt ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
chkopt	Check option
	The option forces the function to check the property handle befor running executing the function. You can pass NO to avoid unnecessary checking.

IsStructure - Is the type of the PropertyHandle a defined Structure ?

The function returns true (YES) if the type of the property handle is a structure and false (NO) if the type of the property handle is a basic type or an enumeration.

```
logical PropertyHandle :: IsStructure ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsText - Check property type for text

The function checks whether the propertyhandle refers to a text field (YES) or not (NO). Text fields are fields with type CHAR, MEMO, STRING and CCHAR.

```
logical PropertyHandle :: IsText (char chkopt ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
chkopt	Check option The option forces the function to check the property handle befor running executing the function. You can pass NO to avoid unnecessary checking.

IsTransient - Is property transient

Attributes as well as references might be defined as transient properties. Transient properties are filled by the application.

```
logical PropertyHandle :: IsTransient ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsTrue - Is value for property TRUE?

The function returns whether the property handle is empty (NO) or not (YES). (see IsEmpty()).

```
logical PropertyHandle :: IsTrue ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsTyped - Is instance typed?

The function returns YES when the property handle is valid and not a VOID or week typed reference. Otherwise the function returns NO.

```
logical PropertyHandle :: IsTyped ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsValid - Check for valid property handle

The function checks whether the property handle is valid. The function does not check whether the handle is opened (see **{.r Check** ()}.

This function should be called when the application is not shure whether the handle is correct or not.

```
logical PropertyHandle :: IsValid (logical topt ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
topt	

IsValidText - Checks text fields for valid characters

The function validates a text according to the characters passed in string. If the property handle does not refer to a text filed or if the text contains other characters than defined in the string the function returns false (NO).

```
logical PropertyHandle :: IsValidText (char *string ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
string	String area Pointer to the 0-terminated string area.

IsWeakTyped - Is reference weak typed

The function returns YES if the reference or collection is defined as weak typed. In this case the type of instances may change from instance to instance. All instances in the reference are bases on a common base structure.

The type of the common base structure can be retrieved with GetStructDef(). The type for the selected instance can be retrieved with GetCurrentType().

Befor inserting a new instance to a weak-typed refererence the type must be set with SetType().

```
logical PropertyHandle :: IsWeakTyped ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

IsWrite - Can instance be updated?

When an instance has been selected for the property handle the function returns whether the instance can be updated (YES) or not. The instance cannot be updated for several reasons:

1. The property handle is opened for read, only

2. The instance is permanently write protected

3. The instance is locked by another user

4. The current applications does not have rights for updateing the instance

5. The instance is an imbedded part of another instance which cannot be updated

```
logical PropertyHandle :: IsWrite ( ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

KeyToString - Convert internal key to string

The passed key will be converted from the internal key instance format into an extended SDF string. As field separator '|' is used. Structure levels are enclosed in '{}'. Normally the key passed is assumed to be structured according to the sort key selected for the property handle or according to the identifying key (when no sort key is defined). It is, however, also possible to pass a valid key name for conversion.

```
char *PropertyHandle :: KeyToString (Key key_string, Key
                        key_val, char *key_name_w )
```

Return value	The key is provided as ESDF key. {} are used as instance parenthesis, | is used as property delimiter. Delimiters may change when defined differently in the DataFormat option.
key_string	String area for key The key is provided as ESDF key. {} are used as instance parenthesis, | is used as property delimiter. Delimiters may change when defined differently in the DataFormat option.
key_val	Internal key value The key value structure corresponds to the structure of the passed or selected key.
key_name_w	Key name for conversion The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks. If no key name is passed (NULL) the sort key according to the selected sort order is used instead,

Locate - Locate object by identity

The function is searching for an instance with the given local identity (LOID) in the collection of the property handle. The function returns NO when the instance could be located. The function returns an error (YES) in case of an error or when the instance is not member of the collection.

When passing YES for read_opt the instance is selected in the property handle. Otherwise it is located, only and can be read with Get(CUR_INSTANCE).

```
logical PropertyHandle :: Locate (int32 obident, logical
                     read_opt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
obident	Instance identity The identity refers to a persistent instance that can be refeenced within the database. Imbedded instances and exclusive base structure instances do not have an instance identity.
read_opt	Read option The option forces the function to read the instance when it could be located.

LocateKey - Locate instance according to key

The instance with the passed key value will be located in the currently selected index. In case of error or when no instance with the given key was found the function returns an error (YES). Otherwise the instance is located and can be read with Get(CUR_INSTANCE).

Passing NO for exact the function tries to locate the instance with the next higher key value.

i00 - Lokate key by key value

This implementation locates the key by means of the key structure passed to the function. The key must be passed according to the structure of the key including trailing blaks.

```
logical PropertyHandle :: LocateKey (Key sort_key, logical exact
                        )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
sort_key	Sort key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey}() function. When no key is passed by the application an empty key (without data area) will be passed.
exact	Exact option The exact option forces the fuction to locate the instance with the exact key. Default: YES

i02 - Lokate instance by property handle

When the property handle contains a string value, the string will be converted into a key for locating the instance. When the property handle refers to a complex instance of the same type or a base type of the current type in the property handle, the instance key in the passed property handle is used for locating the key.

When the property handle contains a numeric value, no instance is located.

```
logical PropertyHandle :: LocateKey (PropertyHandle &prop_hdl,
                         logical exact )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_hdl	Property Handle

	Is a reference to an (usually) opened property handle.
exact	Exact option The exact option forces the fuction to locate the instance with the exact key. Default: YES

LocatePath - Locate path for path collection handle

The function ensures that all property handles in a hierarchy are positioned, i.e. an instance is selected for all upper property handles and the calling handle itself. If no instance is selected in any handle in the hierarchy the function automatically tries to locate the first instance for those property handles.

```
logical PropertyHandle :: LocatePath ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Lock - Lock instance

This function allows locking the selected instance of the property handle within the application. As long as the instance is locked no other user is able to access the instance. Instances for shared base structures are not automatically included in the locking and must be locked separately. Locked instances can be unlocked using the Unlock() function. They are automatically unlocked, when another instance is selected in the property handle.

The function returns NO when the instance has been locked successfully. It returns en error (YES) when the instance is already locked by another application, when no instance is selected in the property handle or when an error occurred.

```
logical PropertyHandle :: Lock ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

LockSet - Lock collection

This function allows locking a collection handle referenced in a collection property handle within the application. As long as the collection is locked no other user is able to access the collection. Locked collection can be unlocked using the UnlockSet() function. The collection is automatically unlocked, when the property handle is closed or another instance is selected in the upper property handle.

The function returns NO when the collection has been locked successfully. It returns en error (YES) when the collection is already locked by another application, when no instance is selected in the upper property handle (when existing) or when an error occurred.

```
logical PropertyHandle :: LockSet ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

MarkUnused - Mark property handle as unused

When referring a property handle recursively this may result in never deleting the resources allocated with the property handle. Recursive references to property handles may happen when setting transient references (SetTransientProperty()) to an upper property handle. In this case the programm has to mark the property handle as unused. Property handles marked as unused must be marked as used befor being closed or replaced (CopyHandle()).

Using this function improperly (e.g. after constructing a property handle) may release the resources allocated to the property handle immediately.

```
void PropertyHandle :: MarkUnused ( )
```

MarkUsed - Mark property handle as used

A property handle should be marked as used before being closed or replaced in a recursive reference.

```
void PropertyHandle :: MarkUsed ( )
```

Minus - Substract collections

The result collection contains all instances that exist in the first but not in the second operand collection. The existence of an instance in a collection can be checked based on the sort key (passing YES for ik_opt) or on local identities (LOID). Using the LOID is save but comparing the key is much faster. Hence, the key check should be used whenever possible.

Calling the function with one operator creates the difference collcetion between the calling (first operand) and the passed collection and stores the result in the calling collection.

Otherwise the operation is performed with the passed operand collectios storing the result in the collection referenced by the calling property handle. When the calling property handle refers to a non empty collection all instances are removed before performing the operation. When the calling property handle is empty the function creates a temporary extend for storing the result.

i0

```
PropertyHandle &PropertyHandle :: Minus (PropertyHandle
                  &prop_hdl1, PropertyHandle &prop_hdl2, char
                  sk_opt )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prop_hdl1	First Property handle Reference to an opened property handle.
prop_hdl2	Second Property handle Reference to an opened property handle.
sk_opt	Sort key option The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances. Default: YES

i1

```
PropertyHandle &PropertyHandle :: Minus (PropertyHandle
                 &prophdl_ref, char sk_opt )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.
sk_opt	Sort key option The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances. Default: YES

Modify - Mark property as modified

The function marks the instance selected in the property handle as modified. This is usually done automatically when assigning a value to a property handle. When, however, writing data directly to the instance the Modify() function must be called to register the modification. Otherwise the modification will not be stored to the database.

The function returns NO when executed successfully. When no instance is selected or in case of an error the function returns an error (YES).

```
logical PropertyHandle :: Modify ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Move - Move instance to another collection

The function moves an instance from the source collection (source_handle) to the collection of the current property handle. This is the only way, to move dependent or owned instances from one collection to another one. It is also a save way to copy instances avoiding duplicate key problems, that may result from the inverse reference for local collections. The function allows also moving instances between distinct sub-collections of an extent.

When the type of source and target instance is the same and when both, source and target property handle have been opened for the same database handle, the instance is removed from the source collection and inserted into the target collection. In this case the instance does not change the local and global identity. When the two collection differ in type, the instance is copied from the source to the target collection and removed/deleted from the source collection afterwards.

When replacing existing instances is required the instance is identified in the target collection by key according to the sort order set for the target collection. When a unique sort order is set and an instance with the same key as the source instance does already exist, the instance is removed/deleted from the target collection before moving the source instance to the target collection.

The function returns an instance handle to the instance selected in the property handle (instance moved). The function returns an empty instance handle, when the it terminates with error.

i00 - Move without rename

The function moves the instance without renaming. When the instance exists in the target collection the function returns an error.

```
Instance PropertyHandle :: Move (PropertyHandle &source_handle,
                    PIREPL replopt )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
source_handle	Source property handle The source property handle must be opened and an instance must be selected in the handle.
replopt	Replace option The replace option controls the behaviour of the copy function. Options that can be used here are: REPL_none - do not replace existing instances REPL_direct - copy attributes, only (but no global identities) REPL_GUID - copy attributes including global identity REPL_local - replace collections owned by the instance REPL_all - replace primary relationships REPL_no_create - copy primary relationships without creating new instances

i01 - Move with rename

The function allows renaming an instance while moving it to the target collection. When the new key does already exist in the target collection the instance is overwritten depending on the replace option.

```
Instance PropertyHandle :: Move (PropertyHandle &source_handle,
                           Key new_key, PIREPL replopt )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
source_handle	Source property handle The source property handle must be opened and an instance must be selected in the handle.
new_key	New key for the instance

	The key passed for renaming the instance must be structured according to the currently selected sort order.
replopt	Replace option The replace option controls the behaviour of the copy function. Options that can be used here are: REPL_none - do not replace existing instances REPL_direct - copy attributes, only (but no global identities) REPL_GUID - copy attributes including global identity REPL_local - replace collections owned by the instance REPL_all - replace primary relationships REPL_no_create - copy primary relationships without creating new instances

MoveDown - Move instance down

In an unordered collection or in a collection ordered by __AUTOIDENT the position of an instance can be moved up or down. Moving the instance down in an unordered position will change the position of the instance, only. Mowing it down in a collction ordered by __AUTOIDENT will update the identifying number of the instance.

The function will not change the position for instances in any other type of collection.

```
logical PropertyHandle :: MoveDown ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

MoveUp - Move instance one position up

In an unordered collection or in a collection ordered by __AUTOIDENT the position of an instance can be moved up or down. Moving the instance up in an unordered position will change the position of the instance, only. Mowing it up in a collction ordered by __AUTOIDENT will update the identifying number of the instance.

The function will not change the position for instances in any other type of collection.

```
logical PropertyHandle :: MoveUp ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

NextKey - Locate next key

The function tries to locate the next key according to the selected order starting from the selected or located instance. When passing a key the instance with this key or the next lower (if not existing) is the starting point. The function locates the key next to the starting point. The function returns the next key located or an empty Key handle in case of an error. When calling the function for indexes allowing duplicate key values, NextKey returns also duplicates.

The instance is located but not selected in the property handle. It cen be selected calling Get(CUR_INSTANCE) after calling NextKey(). Since the function is not reading instances but parsing the index only, it provides fast access to the keys of a collection.

Passing a switch_level allows defining the last key components that must alter. This allows e.g. reading all instances for key duplicates when fixing the lats key component.

```
Key PropertyHandle :: NextKey (Key sort_key_w, int16
                    switch_level )
```

Return value	The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the ({.r pib.StringToKey}()) function. Regardles on the type key values are passed as (char *) areas.
sort_key_w	Sort key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey() function. Default: Key() (empty key)
switch_level	Number of fixed key component The switch level defines the key component number that must not change when calling NextKey beginning with 0 for the first key component. Default: AUTO

NoWrite - Is instance write protected?

The function returns whether the instan can be updated (NO) or not (YES).

(-> IsWrite())

```
logical PropertyHandle :: NoWrite ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

Open - Open property handle

Open property handle opens a property handle for a persistent or transient data source. A data source is a collection, object instance or an elementary database field. A data source contains the data for a property of a specific object. Usually property handles are opened when constructimg them. You can, hawever, create an unopened property handle using the dummy constructor (without parameters) or by closing another property handle.

If the property handle to be opened is a subsequent property handle the parend must be opened. The datasource provided in the subsequent property handle depends on the parents property handle current selection and will be provided automatically whenever the parent property handle changes its current selection.

You can open static property handles for constants or other elementary data sources as well as for structured instances or transient collections using the appropriate open function.

When applying the Open() function to a property handle that has been opened previously the handle is closes implicitely before reopening. You cannot reopen property handles that have been provided with GetPropertyHandle().

When creating a copy handle sort order and selected instance are set in the copy handle as well.

c1 - Open non sharing copy for a property handle

The function opens a copy of the passed property handle with an own cursor and an own instance area.

```
logical PropertyHandle :: Open (const PropertyHandle &cprop_hdl
                         )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

c2 - Open a sharing copy for a property handle

The function opens a copy of the passed property handle that shares the instance area and cursor with its origin.

```
logical PropertyHandle :: Open (PropertyHandle *property_handle
                            )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
property_handle	Pointer to a property handle Is a pointer to an (usually) opened property handle.

d1 - Open subordinated property handle

The function openes a subordinated property handle of the instance. Since the property handlels for instance properties are part of the instance the function creates a copy handle with an own cursor. It behaves, however, like a normal subordinated property handle, that depends on the selection in the upper property handle (if there is any).

```
logical PropertyHandle :: Open (PropertyHandle &prop_hdl, char
                            *prop_path )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.
prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

d2 - Open subordinated property handle

The constructor provides a subordinated property handle of the instance. Since property handlels for an instance are part of the instance the function provides a property handle that shares area and cursor with the property handle in the instance. As subordinated property handle it depends on the selection in the upper property handle (if there is any).

```
logical PropertyHandle :: Open (PropertyHandle *property_handle,
                          char *prop_path )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
property_handle	Pointer to a property handle Is a pointer to an (usually) opened property handle.
prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

d3 - Open extent property handle

The function opens a property handle for a global collection (extent). The extent name passed may contain symbolic references to system variables (e.g. "%EXT_PREF%Pers") which are resolved according to the current setting of the referenced system variables. A key name can be passed to set the sort order for the property handle. If no key is passed the sort order is set to the default order.

You may open a transient extent that stores data only in main storage by passing the transient_w option.

```
logical PropertyHandle :: Open (const DBObjectHandle
                          &obhandle_refc, char *extnames, PIACC accopt,
                          logical transient_w, char *key_name_w )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

obhandle_refc	Const reference to database object handle The reference refers to an opened or not opened database object handle.
extnames	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).
transient_w	Transient option To create transient property handles transient=YES has to be passed. In this case the property handle instences and indexes are stored in main storage, only. Default: NO
key_name_w	Key name for conversion The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks. If no key name is passed (NULL) the sort key according to the selected sort order is used instead,

d5 - Open property handle for temporary extent

The function opens a property handle for a temporary extent for storing results of set operations. Temporary extents are stored in a separate temporary file and are available only as long as the process runs.

```
logical PropertyHandle :: Open (const DBObjectHandle
                 &obhandle_refc, char *strnames, char *keyname,
                 char *baseexts_w, logical weak_opt_w, logical
                 own_opt_w )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
obhandle_refc	Const reference to database object handle The reference refers to an opened or not opened database object handle.

strnames	Structure name The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.
keyname	Name of sort key The order key name must be a key name defined for the given structure. The sort key is passed as 0-terminated string with maximum 40 characters.
baseexts_w	Name for base extent A base extent or base collection can be passed that defines a superset for the temporary extent. The extent name is passed as 0-terminated string with maximum 40 characters.
weak_opt_w	Weak-typed option This option must be true (YES) when a collection may refer to instances of differet types, wich are based on the same base structure.
own_opt_w	Owning collection This option must be set to true (YES) if the collection owns the instances it is referring to. In this case the collection may not refer to instances from other collections. Removing instances from an owning collection will result in deleting the instance completely.

d8 - Open a view property handle

The function opens a view property handle based on the view definition passed to the function. The view is opened relatively to the property handle passed as prop_hdl or as view in a global context when passing an empty property handle. The view can be opened in read, update or write mode (accopt).

```
logical PropertyHandle :: Open (const DBObjectHandle
                 &obhandle_refc, DBViewDef &view_def, Proper-
                 tyHandle &prop_hdl, PIACC accopt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

obhandle_refc	Const reference to database object handle The reference refers to an opened or not opened database object handle.
view_def	View definition A view definition defines the elements ans selection condition for a view.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).

d9 - Open operation property handle

Operation property handles can be opened in order to define a set operation as sub-ordinated property handle. Operation property handles can be used within path properties acting like a normal property handle, except that one cannot create new instances. Depending to the operation type operation property handle pass instance by instance (e.g. where) or do calculate the complete result set before passing the first instance.

```
logical PropertyHandle :: Open (OperationTypes operation_type,
                  PropertyHandle &prop_hdl, char sk_opt, logical
                  distinct, char *rule )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
operation_type	Operation type

	The operation type describes the set operation to be performed in a view or operational path. When referring to operations the following property names should not be used, since they are interpreted as operations (not case sensitive):
	select, define
	having, where
	group_by, group
	order, order_by
	from
	minus
	intersect
	join
	update
prop_hdl	Property Handle
	Is a reference to an (usually) opened property handle.
sk_opt	Sort key option
	The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances.
	Default: YES
distinct	Distinct option
	Passing a distinct option YES forces the function to remove duplicates from the result collection.
rule	OPeration rule
	Depending on the operation type the operation rule describes the details. Unsually, the rule is provided as ODABA OQL expression (where, group), but other formats are possible as well.

d9a - Open operation property handle (top)

Operation property handles can be opened in order to define a set operation. As top-handle, the number of setoperations, that can be defind, is limited to FROM (product). Operation property handles can be used within path properties acting like a normal property handle. The top operation property handle passes instance by instance on request. Sequential forward access is the most efficient one.

```
logical PropertyHandle :: Open (OperationTypes operation_type,
                 DBObjectHandle &object_handle, char sk_opt,
                 logical distinct, char *rule )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
operation_type	Operation type The operation type describes the set operation to be performed in a view or operational path. When referring to operations the following property names should not be used, since they are interpreted as operations (not case sensitive): select, define having, where group_by, group order, order_by from minus intersect join update
object_handle	Database Object handle This is a pointer to an opened Database Object handle.
sk_opt	Sort key option

The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances.

Default: YES

distinct	Distinct option Passing a distinct option YES forces the function to remove duplicates from the result collection.
rule	OPeration rule Depending on the operation type the operation rule describes the details. Unsually, the rule is provided as ODABA OQL expression (where, group), but other formats are possible as well.

i01 - Open subordinated property handle

The function opens a subordinated property handle for an unbound data instance (a data instance that is not connected to the database). The property handle has no connection to the database and does not support database access functions.

The data area is the data area of the property in the instance passed to the function. If no instance is passed, no data area is allocated. This can be done later using the SetInstance() function.

```
logical PropertyHandle :: Open (DBStructDef *struct_def, char
                          *prop_names, char *instance )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
struct_def	Pointer to generel structure definition The structure definition (DBStructDef) contains the metadata for the instance, i.e. information for the structure and its properties.
prop_names	Property name The property name is passed as 0-terminated string. It may contain a property path that consists of a sequence of property names.

instance	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

i11 - Open an unbound property handle

The function opens an unbound property handle according to the field definition (field_def) passed to the constructor. An initial value can be passed as string value to initialize the data area allocated for the property handle.

```
logical PropertyHandle :: Open (Dictionary *dictptr, DBFieldDef
                          *field_def, char *init_string, logical in-
                          it_opt, logical const_opt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dictptr	Dictionary handle An opened dictionary handle is passed.
field_def	Property definition The property defintion contains the metadata for the referenced property instance..
init_string	Initial value The initial value for the property is passed as 0-terminated string.
init_opt	Initialize option
const_opt	Constant Option Defines a property handle as constant.

i12 - Open an unbound property handle

The function opens an unbound property handle according to the type passed in typenames and the properties passsed to the function. An initial value can be passed as string value to initialize the data area allocated for the property handle.

```
logical PropertyHandle :: Open (Dictionary *dictptr, char
                 *prop_names, char *typenames, SDB_RLEV
                 ref_level, uint16 size, uint16 precision,
                 uint16 dimension, char *init_string )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dictptr	Dictionary handle An opened dictionary handle is passed.
prop_names	Property name The property name is passed as 0-terminated string. It may contain a property path that consists of a sequence of property names.
typenames	Type name The type name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
ref_level	Reference level The reference level describes the way and the level of instance references.
size	Size Size of the instance or property area.
precision	Precision The precision defines the number of decimal positions behind the decimal point for numerical valued. For date and time values it defines the way of presenting the values in charachter presentations.
dimension	Dimension

The dimension describes the property dimension. this is the maximum number of instances that can be stored for the property. The function returns 0 (UNDEF) if there is no limit (collection) or the dimension (cardinality) defined for the property.

init_string	Initial value The initial value for the property is passed as 0-terminated string.

i13 - Open an unbound property handle with type name

The function opens an unbound property handle according to the type passed in typenames.

```
logical PropertyHandle :: Open (Dictionary *dictptr, char
                          *typenames )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dictptr	Dictionary handle An opened dictionary handle is passed.
typenames	Type name The type name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

i14 - Opening an unbound property handle with database definition

The function opens an unbound property handle according to the dictionary SDB_Member definition passed to the function (dbmptr).

```
logical PropertyHandle :: Open (DictionaryHandle &dict_handle,
                          SDB_Member *dbmptr )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dict_handle	Dictionary handle

	The dictionary handle usually refers to an opened dictionary. To check whether a dictionary is opened you can use the !-operator.
dbmptr	Member definition

i20 - Open an unbound property handle with structure definition

The function opens an unbound property handle for an unbound data instance (a data instance that is not connected to the database). The property handle has no connection to the database and does not support database access functions.

The instance passed to the function (instance) is set as instance area for the property handle, i.e. the handle shares the data area with the application.

```
logical PropertyHandle :: Open (DBStructDef *strdef, char
                  *instance )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
strdef	Structure definition The structure definition is provided in the internal format as pointer to a DBStructDef object.
instance	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

i21 - Open an unbound property handle

The function opens an unbound property handle according to the field definition (field_def) passed to the constructor. An initial value can be passed according to the type of the property handle to initialize the data area allocated for the property handle.

```
logical PropertyHandle :: Open (DBFieldDef *field_def, Instance
                  initinst )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
field_def	Property definition The property defintion contains the metadata for the referenced property instance..
initinst	Initializing instance Instance for initializing the instance area for the property handle.

u1 - Open property handle for a 32-bit integer value

The function opens an unbound property handle for a platform independent 32-bit integer value (int32).

```
logical PropertyHandle :: Open (int32 int_val )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
int_val	Integer value

u2 - Open property handle for a string value

The function opens an unbound property handle for a string value (STRING). The area is allocated with the size of the string passed to the constructor. The string is copied into the instance area owned by the property handle. To enable dynamical resize featur for the property handle use the SetDynLength() function.

```
logical PropertyHandle :: Open (char *string )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
string	String area Pointer to the 0-terminated string area.

u3 - Open property handle for a string value

The function opens an unbound property handle for a string value (STRING). The area is set to the string pointer passed to the function.

```
logical PropertyHandle :: Open (char *string, int32 string_len )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
string	String area Pointer to the 0-terminated string area.
string_len	String length The string length defines the maximum number of characters that can be stored in the string area without counting the terminating 0. Usually this value is 1 less that the allocated string area.

u4 - Open property handle for a double value

The function opens an unbound property handle for a double value (REAL).

```
logical PropertyHandle :: Open (double dbl_value )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbl_value	Double value

u5 - Open property handle for a date value

The function opens an unbound property handle for a date value (DATE).

```
logical PropertyHandle :: Open (dbdt date_val )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
date_val	Date value

The data value is passed in the internal data format.

u6 - OPen property handle for a time value

The function opens an unbound property handle for a time value (TIME).

```
logical PropertyHandle :: Open (dbtm time_val )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
time_val	Time value The time value is passed in the internal data format.

u7 - Open property handle for a date/time value

The function opens an unbound property handle for a date/time value (DATETIME).

```
logical PropertyHandle :: Open (dttm datetime_val )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
datetime_val	Date-Time value A date-time value or time point is passed in the internal date-time format.

u8 - Open property handle for a logical value

The function opens an unbound property handle for a logical value (LOGICAL).

```
logical PropertyHandle :: Open (logical logval )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
logval	Logical value Is a logical (bool) value.

x1 - Open an undefined property handle

The function opens an undefined and unbound property handle. Before using the property handle definition and instance area must be set.

```
logical PropertyHandle :: Open ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

OpenAccessPath - Open Access Path

An access path is a complex expression, which allows defining an extended view to the database. This includes the features of SQL and OQL, which are also included in the ODABA View definition, but adds some more ODABA specific facilities. Details for defining an access path are described in 'ODABA User's Guide'.

After Opening an acess path it is not inilialized, i.e. metadata is not yet available. The access path will be initialized whwn calling the Get() or ToTop() function.

Accesspathes are used read only. Only in some special cases you may open the access path in update mode.

i00

```
logical PropertyHandle :: OpenAccessPath (PropertyHandle
                 &prop_hdl, BNFData *parm_data )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i01

```
logical PropertyHandle :: OpenAccessPath (PropertyHandle
                 &prop_hdl, ACObject *obhandle, BNFData
                 *parm_data )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.
obhandle	Database Object Handle This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.

i02

```
logical PropertyHandle :: OpenAccessPath (PropertyHandle
                    &prop_hdl, char *path_prop )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i03

```
logical PropertyHandle :: OpenAccessPath (PropertyHandle
                    &prop_hdl, ACObject *obhandle, char *path_prop
                    )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.
obhandle	Database Object Handle This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.

i04

```
logical PropertyHandle :: OpenAccessPath (ACObject *obhandle,
                          BNFData *parm_data, PIACC access_mode )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
obhandle	Database Object Handle This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.
access_mode	Access mode The access option defines the way instances in a property handle are to be accessed (read, update, write). Default: PI_Read

i05

```
logical PropertyHandle :: OpenAccessPath (ACObject *obhandle,
                          char *path_prop, PIACC access_mode )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
obhandle	Database Object Handle This is the database object handle or the database handle when referring to the root object or the dictionary handle when referring to the root object of the dictionary database.
access_mode	Access mode The access option defines the way instances in a property handle are to be accessed (read, update, write). Default: PI_Read

OpenHierarchy - Open hierarchy property handle

The function opens a hierarchy property handle, which creates a property handle for bottom property handle. The function will duplicate the complete hierarchy including the top property handle. The function copies the selectins from the source hierarchy, i.e. the result contains the same selected instance as the source hierarchy.

The function returns a property handle for the collection in the bottom property handle. Upper parent property handle, which have been created by the function, will be destructed automatically when destroying the bottom property handle. When no top property handle is passed or when the top property handle is not part of the source hierarchy, the returned hierarchy ends with the extent property handle for the bottom property handle.

You may access upper property handles by referring parent property handle with the GetParentProperty() function. Since all parents are copies, you may change the selection in any parent handle without danger.

When passing true (YES) for path option, the function will turn the property hierarchy into a path property, which automatically iterates on higher levels.

When the source handle is opened in write mode or when any of the property handles in the hierarchy are opened in write mode, the copy might refer to a write protected instance. You can use the Refresh() function to remove the write protection, after instances have been released in the other hierarchy.

The root for the hierarchy it the top handle of the hierarchy. Since this is a copy handle, the complete hierarchy will be closed, when closing the origin handle (or the extend node) implicitly or explicitly.

```
logical PropertyHandle :: OpenHierarchy (PropertyHandle
                  *bottom_ph, PropertyHandle *top_ph, logical
                  path_opt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
bottom_ph	Lowest property handle

This is a pointer to the lowest property handle in a hierarchy or property path.

top_ph	Highest property handle This is a pointer to the top property hande in a hierarchy or property path. When the pointer is NULL, the top property handle is the extent, which is the root of the path.
path_opt	Path oprtion The option indicates, that a peth property will be ctreated.

OwnsData - Owns data area

The function returns whether the property handle owns its data area YES) or whether the area is shared with another property handle (NO).

```
logical PropertyHandle :: OwnsData ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

Position - Select an instance relative to the current selection

The function allow selecting an instance relatively to the currently selected instance. Thus the function allows selecting the next instance in the collection as Position(1) or the previous instance as Position(-1).

The function returns NO when the instance could be selected. When no instance could be selected or an error occurred the function returns an error (YES).

```
logical PropertyHandle :: Position (int16 count )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
count	Number of entries

PositionTop - Position parent collections

The function selectes an instance (first instance) for all upper property handles that are not positioned. The selection for the property handle itself remains unchanged.

```
logical PropertyHandle :: PositionTop ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Power - Raise to power of

The power function is supported for numerical data, only. If passed property handle is not numerical the function tries to convert it into a numerical value. If no conversion is possible the operation fails. The function calculates the value of the property handle raised to the power of the passed value.

The result is returned in a property handle that is created temporarily. You can assign the result to another property handle or performing further operations.

```
PropertyHandle PropertyHandle :: Power (PropertyHandle &prop_hdl
                  )
```

Return value	
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

PropertyHandle - Constructor

The property handle constructor creates and openes a property handle for a persistent or transient data source. A data source is a collection, object instance or an elementary database field. A data source contains the data for a property of a specific object. Except for the dummy constructor (no parameter) property handle are opened when being constructed sucessfully. To check teh success you can use the IsValid() or Check() function.

Cunstructing a subsequent property handle (passing the parent property handle and the property name to the constructor) the parend must be opened. The datasource provided in the subsequent property handle depends on the parents property handle current selection and will be provided automatically whenever the parent property handle changes its current selection.

You can create static property handles for constants or other elementary data sources as well as for structured instances or transient collections using the appropriate constructur function.

c1 - Create non sharing copy for a property handle

The constructor creates a copy of the passed property handle with an own cursor and an own instance area.

```
PropertyHandle :: PropertyHandle (const PropertyHandle &cprop_hdl )
```

cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

c2 - Create a sharing copy for a property handle

The constructor creates a copy of the passed property handle that shares the instance area and cursor with its origin.

```
PropertyHandle :: PropertyHandle (PropertyHandle *property_handle )
```

property_handle	Pointer to a property handle Is a pointer to an (usually) opened property handle.

d0 - Create copy handle

The constructor creates a copy property handle that shares area and cursor with the handle passed as nodeptr.

```
PropertyHandle :: PropertyHandle (node *nodptr )
```

nodptr

d1 - Create subordinated property handle

The constructor provides a subordinated property handle of the instance. Since the property handlels for instance properties are part of the instance the function creates a copy handle with an own cursor. It behaves, however, like a normal subordinated property handle, that depends on the selection in the upper property handle (if there is any).

```
PropertyHandle :: PropertyHandle (PropertyHandle &prophdl_ref, char *prop_names )
```

prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.
prop_names	Property name The property name is passed as 0-terminated string. It may contain a property path that consists of a sequence of property names.

d2 - Create subordinated property handle

The constructor provides a subordinated property handle of the instance. Since property handlels for an instance are part of the instance the function provides a property handle that shares area and cursor with the property handle in the instance. As subordinated property handle it depends on the selection in the upper property handle (if there is any).

```
PropertyHandle :: PropertyHandle (PropertyHandle *property_handle, char *prop_path )
```

property_handle	Pointer to a property handle Is a pointer to an (usually) opened property handle.
prop_path	Property path

The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

d3 - Create extent property handle

The constructor creates a property handle for a global collection (extent). The extent name passed may contain symbolic references to system variables (e.g. "%EXT_PREF%Pers") which are resolved according to the current setting of the referenced system variables.

```
PropertyHandle :: PropertyHandle (const DBObjectHandle &obhandle_refc, char *extnames, PIACC accopt )
```

obhandle_refc	Const reference to database object handle The reference refers to an opened or not opened database object handle.
extnames	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).

d4 - Create extent property handle

The constructor creates a property handle for a global collection (extent). The extent name passed may contain symbolic references to system variables (e.g. "%EXT_PREF%Pers") which are resolved according to the current setting of the referenced system variables. A key name can be passed to set the sort order for the property handle. If no key is passed (NULL) the sort order is set to the default order.

```
PropertyHandle :: PropertyHandle (const DBObjectHandle &obhandle_refc, char *extnames, char *keynames, PIACC accopt, logical transient_w )
```

obhandle_refc	Const reference to database object handle

	The reference refers to an opened or not opened data-base object handle.
extnames	Extent name The extent name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
keynames	Key name The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.
accopt	Access option The access option defines the way instances in a prop-erty handle are to be accessed (read, update, write).
transient_w	Transient option To create transient property handles transient=YES has to be passed. In this case the property handle instences and indexes are stored in main storage, only. Default: NO

d5 - Create property handle for temporary extent

The constructor creates a property handle for a tempo-rary extent for storing results of set operations. Tempo-rary extents are stored in a separate temporary file and are available only as long as the process runs.

```
     PropertyHandle :: PropertyHandle (const
DBObjectHandle &obhandle_refc, char *strnames,
char *keyname, char *baseexts_w, logical
weak_opt_w, logical own_opt_w )
```

obhandle_refc	Const reference to database object handle The reference refers to an opened or not opened data-base object handle.
strnames	Structure name The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trail-ing blanks.
keyname	Name of sort key

The order key name must be a key name defined for the given structure. The sort key is passed as 0-terminated string with maximum 40 characters.

baseexts_w	Name for base extent A base extent or base collection can be passed that defines a superset for the temporary extent. The extent name is passed as 0-terminated string with maximum 40 characters.
weak_opt_w	Weak-typed option This option must be true (YES) when a collection may refer to instances of differet types, wich are based on the same base structure.
own_opt_w	Owning collection This option must be set to true (YES) if the collection owns the instances it is referring to. In this case the collection may not refer to instances from other collections. Removing instances from an owning collection will result in deleting the instance completely.

d6 - Create a global view property handle

The constructor creates a view property handle based on the view definition passed to the function. The view is opened in a global context. The view can be opened in read, update or write mode (accopt).

```
PropertyHandle :: PropertyHandle (const
DBObjectHandle &obhandle_refc, DBViewDef
&view_def, PIACC accopt )
```

obhandle_refc	Const reference to database object handle The reference refers to an opened or not opened database object handle.
view_def	View definition A view definition defines the elements ans selection condition for a view.
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).

d7 - Create a relative view property handle

The constructor creates a view property handle based on the view definition passed to the function. The view is opened relatively to the property handle passed as prop_hdl. The view can be opened in read, update or write mode (accopt).

```
        PropertyHandle :: PropertyHandle
(DBViewDef &view_def, PropertyHandle
&prop_hdl, PIACC accmode )
```

view_def	View definition A view definition defines the elements ans selection condition for a view.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.
accmode	Access mode The access option defines the way instances in a property handle are to be accessed (read, update, write).

d8 - Create a relative or global view property handle

The function opens a view property handle based on the view definition passed to the function. The view is opened relatively to the property handle set in the view definiition or as view in a global context when this property handle is empty. The view can be opened in read, update or write mode (accopt).

```
        PropertyHandle :: PropertyHandle
(DBViewDef &view_def, PIACC accmode )
```

view_def	View definition A view definition defines the elements ans selection condition for a view.
accmode	Access mode The access option defines the way instances in a property handle are to be accessed (read, update, write).

d9 - Operation Property Handle

Operation property handles can be created in order to define a set operation as sub-ordinated property handle. Operation property handles can be used within path properties acting like a normal property handle. Depending to the operation type operation property handle pass instance by instance (e.g. where) or do calculate the complete result set before passing the first instance. Sequential forward access is the most efficient one.

```
     PropertyHandle :: PropertyHandle (Opera-
tionTypes operation_type, PropertyHandle
&prop_hdl, char sk_opt, logical distinct, char
*rule )
```

operation_type — Operation type

The operation type describes the set operation to be performed in a view or operational path. When referring to operations the following property names should not be used, since they are interpreted as operations (not case sensitive):

select, define

having, where

group_by, group

order, order_by

from

minus

intersect

join

update

prop_hdl — Property Handle

Is a reference to an (usually) opened property handle.

sk_opt — Sort key option

The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances.

Default: YES

distinct — Distinct option

Passing a distinct option YES forces the function to remove duplicates from the result collection.

rule	OPeration rule Depending on the operation type the operation rule describes the details. Unsually, the rule is provided as ODABA OQL expression (where, group), but other formats are possible as well.

d9a - Operation Property Handle

Operation property handles can be created in order to define a set operation. As top-handle, the number of setoperations, that can be defind, is limited to FROM (product). Operation property handles can be used within path properties acting like a normal property handle. The top operation property handle passes instance by instance on request. Sequential forward access is the most efficient one.

```
      PropertyHandle :: PropertyHandle (Opera-
tionTypes operation_type, DBObjectHandle
&object_handle, char sk_opt, logical distinct,
char *rule )
```

operation_type	Operation type The operation type describes the set operation to be performed in a view or operational path. When referring to operations the following property names should not be used, since they are interpreted as operations (not case sensitive): select, define having, where group_by, group order, order_by from minus intersect join update
object_handle	Database Object handle

	This is a pointer to an opened Database Object handle.
sk_opt	Sort key option The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances. Default: YES
distinct	Distinct option Passing a distinct option YES forces the function to remove duplicates from the result collection.
rule	OPeration rule Depending on the operation type the operation rule describes the details. Unsually, the rule is provided as ODABA OQL expression (where, group), but other formats are possible as well.

i01 - Create subordinated property handle

The constructor creates a subordinated property handle for an unbound data instance (a data instance that is not connected to the database). The property handle has no connection to the database and does not support database access functions.

The data area is the data area of the property in the instance passed to the function. If no instance is passed, no data area is allocated. This can be done later using the SetInstance() function.

```
        PropertyHandle :: PropertyHandle
(DBStructDef *strdef, char *prop_names, In-
stance instance_w )
```

strdef	Structure definition The structure definition is provided in the internal format as pointer to a DBStructDef object.
prop_names	Property name The property name is passed as 0-terminated string. It may contain a property path that consists of a sequence of property names.
instance_w	Instance area

Instances do have the type of the referenced property handle (collection type). The instance contains a reference to a propertly structured area.

You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

Default: Instance()

i11 - Create unbound property handle

The constructor creates an unbound property handle according to the field definition (field_def) passed to the constructor. An initial value can be passed as string value to initialize the data area allocated for the property handle.

```
      PropertyHandle :: PropertyHandle (Dic-
tionary *dictptr, DBFieldDef *field_def, char
*init_string, logical init_opt, logical
const_opt )
```

dictptr	Dictionary handle An opened dictionary handle is passed.
field_def	Property definition The property defintion contains the metadata for the referenced property instance..
init_string	Initial value The initial value for the property is passed as 0-terminated string.
init_opt	Initialize option
const_opt	Constant Option Defines a property handle as constant.

i12 - Create an unbound property handle

The constructor creates an unbound property handle according to the type passed in typenames and the properties passsed to the function. An initial value can be passed as string value to initialize the data area allocated for the property handle.

```
PropertyHandle :: PropertyHandle (Dictionary *dictptr, char *prop_names, char *typenames, SDB_RLEV ref_level, uint16 size, uint16 precision, uint16 dimension, char *init_string )
```

dictptr	Dictionary handle An opened dictionary handle is passed.
prop_names	Property name The property name is passed as 0-terminated string. It may contain a property path that consists of a sequence of property names.
typenames	Type name The type name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.
ref_level	Reference level The reference level describes the way and the level of instance references.
size	Size Size of the instance or property area.
precision	Precision The precision defines the number of decimal positions behind the decimal point for numerical valued. For date and time values it defines the way of presenting the values in charachter presentations.
dimension	Dimension The dimension describes the property dimension. this is the maximum number of instances that can be stored for the property. The function returns 0 (UNDEF) if there is no limit (collection) or the dimension (cardinality) defined for the property.
init_string	Initial value The initial value for the property is passed as 0-terminated string.

i13 - Create an unbound property handle with type name

The constructor creates an unbound property handle according to the type passed in typenames.

```
PropertyHandle :: PropertyHandle (Dic-
tionary *dictptr, char *typenames )
```

dictptr	Dictionary handle An opened dictionary handle is passed.
typenames	Type name The type name is passed as 0-terminated string or as buffer with trailing blanks and a maximum length of 40 characters.

i14 - Create an unbound property handle with database definition

The constructor creates an unbound property handle according to the dictionary SDB_Member definition passed to the function (dbmptr).

```
PropertyHandle :: PropertyHandle (Dic-
tionaryHandle &dict_handle, SDB_Member *dbmptr
)
```

dict_handle	Dictionary handle The dictionary handle usually refers to an opened dictionary. To check whether a dictionary is opened you can use the !-operator.
dbmptr	Member definition

i20 - Create an unbound property handle with structure definition

The constructor creates an unbound property handle for an unbound data instance (a data instance that is not connected to the database). The property handle has no connection to the database and does not support database access functions.

The instance passed to the function (instance) is set as instance area for the property handle, i.e. the handle shares the data area with the application.

```
PropertyHandle :: PropertyHandle
(DBStructDef *strdef, Instance instance_w )
```

strdef	Structure definition

	The structure definition is provided in the internal format as pointer to a DBStructDef object.
instance_w	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a reference to a propertly structured area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not. Default: Instance()

i21 - Create an unbound property handle

The constructor creates an unbound property handle according to the field definition (field_def) passed to the constructor. An initial value can be passed according to the type of the property handle to initialize the data area allocated for the property handle.

```
PropertyHandle :: PropertyHandle
(DBFieldDef *field_def, Instance initinst )
```

field_def	Property definition The property defintion contains the metadata for the referenced property instance..
initinst	Initializing instance Instance for initializing the instance area for the property handle.

u1 - Create property handle for an integer value

The constructor creates an unbound property handle for an integer value (INT).

```
PropertyHandle :: PropertyHandle (int32
int_val )
```

int_val	Integer value

u2 - Create property handle for a string value

The constructor creates an unbound property handle for a string value (STRING). The area is allocated with the size of the string passed to the constructor. The string is copied into the instance area owned by the property handle. To enable dynamical resize featur for the property handle use the SetDynLength() function.

```
PropertyHandle :: PropertyHandle (char *string )
```

string	String area Pointer to the 0-terminated string area.

u3 - Create property handle for a string value

The constructor creates an unbound property handle for a string value (STRING). The area is set to the string pointer passed to the function.

```
PropertyHandle :: PropertyHandle (char *string, int32 string_len )
```

string	String area Pointer to the 0-terminated string area.
string_len	String length The string length defines the maximum number of characters that can be stored in the string area without counting the terminating 0. Usually this value is 1 less that the allocated string area.

u4 - Create property handle for a double value

The constructor creates an unbound property handle for a double value (REAL).

```
PropertyHandle :: PropertyHandle (double dbl_value )
```

dbl_value	Double value

u5 - Create property handle for a date value

The constructor creates an unbound property handle for a date value (DATE).

```
                 PropertyHandle :: PropertyHandle (dbdt
      date_val )
```

date_val	Date value The data value is passed in the internal data format.

u6 - Create property handle for a time value

The constructor creates an unbound property handle for a time value (TIME).

```
                 PropertyHandle :: PropertyHandle (dbtm
      time_val )
```

time_val	Time value The time value is passed in the internal data format.

u7 - Create property handle for a date/time value

The constructor creates an unbound property handle for a date/time value (DATETIME).

```
                 PropertyHandle :: PropertyHandle (dttm
      datetime_val )
```

datetime_val	Date-Time value A date-time value or time point is passed in the internal date-time format.

u8 - Create property handle for a logical value

The constructor creates an unbound property handle for a logical value (LOGICAL).

```
                 PropertyHandle :: PropertyHandle (logi-
      cal logval )
```

logval	Logical value Is a logical (bool) value.

x1 - Dummy constructor

The function creates an empty property handle without handle pointer.

```
PropertyHandle :: PropertyHandle (
)
```

ProvGenAttribute - Provide generic attributes for new instance

When reading an instance containing generic attributes the generic attributes according to the selected type are provided in the instance only when already existing, i.e. reading an instance will not create missing generic attributes. To provide generic attributes in any case this function can be called that creates missing generic attributes for the read instance.

```
logical PropertyHandle :: ProvGenAttribute ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Provide - Provide instance

The Provide() function allows selecting an instance in a property handle collection by key or position that must not necessarily exist in the collection. The provide function checks whether the instance exists in the collection (Get()). If not existing the instance is created in the collection (-> Add()) and selected.

i0 - Provide instance at position

The function tries to provide an instance at a certain position. When no instance exist at the location passed in set_pos_w, the function creates an instance by position (-> Add(): "Create Instance at position").

```
Instance PropertyHandle :: Provide (int32 set_pos0_w )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
set_pos0_w	Position in collection

The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance.

For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored.

Special positions are

CUR_INSTANCE

CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance.

FIRST_INSTANCE

FIRST_INSTANCE (0) refers to the first instance in a collection according to the selected index (sort order).

LAST_INSTANCE

FIRST_INSTANCE (0) refers to the last instance in a collection according to the selected index (sort order).

i03 - Provide instance by character key

This function is supported because of compatibility reasons and operates as the "Provide instance by key value" function.

```
Instance PropertyHandle :: Provide (char *charkey )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
charkey	Key area The key area is structured according to the key definition (key smcb).

i05 - Provide instance by property

The function checks whether the property handle passes a numerical value or not. When passing a numerical value the function provides an instance at the position according to the number passed in the property handle (-> "Provide instance at position"). When the property handle contains text data, the value in the property handle is interpreted as string key, which will be converted into key and provides an instance by key value (-> "Provide instance by key value").

When the property handle refers to a complex instance of the same type or a base type of the current type in the property handle, the instance key in the passed property handle is used for locating the key.

```
Instance PropertyHandle :: Provide (PropertyHandle &prop_hdl )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i1 - Provide intsance by key value

The function tries to provide an instance with the passed sort key (or ident-key for unordered collections). When no instance exist with the key passed in sort_key, the function creates an instance by key value (-> Add(): "Add Instance by key value").

```
Instance PropertyHandle :: Provide (Key sort_key )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
sort_key	Sort key value

The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey}() function. When no key is passed by the application an empty key (without data area) will be passed.

i2 - Provide intsance by instance

The function extracts the sort key (for ordered collections) or the ident-key (for unordered collections) and tries to locate the instance in the collection with the extracted key value. When no such instance exists the function adds an instance to the collection (-> Add(): "Add instance").

```
Instance PropertyHandle :: Provide (Instance initinst )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
initinst	Initializing instance Instance for initializing the instance area for the property handle.

ProvideArea - Provide instance area

The function provides the instance area for a selected property instance. When no instance is selected in the upper property handle or when no instance is selected in a collection handle the function returns an empty instance.

```
Instance PropertyHandle :: ProvideArea ( ) const
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

ProvideGUID - Provide Global Instance Identifier (GUID)

Usually the GUID is generated when the insatnce is created. It is, however, also possibbe to create GUIDs on demand by not setting autogenerating GUIDs in the structure or collection definition. This function allows explicitely generating a GUID as long as no GUID has been generated for the instance. When the ID already exists, the function returns the current GUID without generating a new one.

For building a GUID for a structure instance the structure must be derived from __OBJECT. Since ProvideGUID is locking the __OBJECT extent it should not be used in long transactions.

```
char *PropertyHandle :: ProvideGUID ( )
```

Return value	The global instance identifier is passed as 0-terminated string with a maximum length of 40 characters.

ProvideGlobal - Provide instance outside the transaction

The function works the same way as the Provide() function, except that global instances are created outside the transaction when not yet existing. When not running in a transactions the function works the same way as Add().

Creating global instances in a transaction prevents all other users from creating global instances for the same extent until the transaction is closed, since the index for the global collection is locked until terminating the transaction. Especially when creating instances via local collections that are based on global collections (extents) uncomfortable locks may block the system. In this case ProvideGlobal() should be used instead of Provide().

Using ProvideGlobal() for creating a new instance the instance will resist in the global collection also when rolling back the transaction.

i0

```
Instance PropertyHandle :: ProvideGlobal (int32 set_pos0_w )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
set_pos0_w	Position in collection The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance. For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored. Special positions are **CUR_INSTANCE** CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance. **FIRST_INSTANCE** FIRST_INSTANCE (0) refers to the first instance in a collection according to the selected index (sort order). **LAST_INSTANCE** FIRST_INSTANCE (0) refers to the last instance in a collection according to the selected index (sort order).

i03

```
Instance PropertyHandle :: ProvideGlobal (char *charkey )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
charkey	Key area The key area is structured according to the key definition (key smcb).

i05

```
Instance PropertyHandle :: ProvideGlobal (PropertyHandle
                    &prop_hdl )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i1

```
Instance PropertyHandle :: ProvideGlobal (Key sort_key )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
sort_key	Sort key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey}() function. When no key is passed by the application an empty key (without data area) will be passed.

i2

```
Instance PropertyHandle :: ProvideGlobal (Instance initinst )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
initinst	Initializing instance Instance for initializing the instance area for the property handle.

ProvideOperation - Provide operation handle

To avoid reopening of an operation handle for the same expression you can search for an operation handle by using this function. The function returns the operation handle for the given expression if it has been ceated already. If not the function creates an operation handle.

```
OperationHandle *PropertyHandle :: ProvideOperation (char
                   *expression )
```

Return value	
expression	OQL expression

An OQL expression defines a condition according to the OQL syntax. OQL expressions must always terminate with ';'. The OQL-Expression is passed as 0-terminated string.

ReadBuffer - Fill instance buffer from position

The function explicitly fills the buffer instances. the position of the first instance to be read is passed in set_pos0. The function removes all buffer instances located in the buffer and refills the buffer. Passing CURRENT_INSTANCE as next position (default) the buffer reads instances beginning with the current position which has been set by the last Get() or LocateKey() function. When no instance is selected the buffer is filled beginning with the first instance.

When reading the last instances in a collection the buffer might not be filled completely. The number of instances read into the buffer is returned from the function.

```
int16 PropertyHandle :: ReadBuffer (int32 set_pos0, int16 direc-
                   tion )
```

Return value	
set_pos0	Position in collection

The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance.

For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored.

Special positions are

CUR_INSTANCE

CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance.

FIRST_INSTANCE

FIRST_INSATNCE (0) refers to the first instance in a collection according to the selected index (sort order).

LAST_INSTANCE

FIRST_INSATNCE (0) refers to the last instance in a collection according to the selected index (sort order).

direction

Refresh - Refresh selected instance

The function checks whether the parent of the property handle is positioned. If not, the function trys to position the parent hierarchy (PositionTop()).

If the parent handle is positioned and the property is an active property the function generates a server event (Refresh).

```
logical PropertyHandle :: Refresh ( )
```

Return value

The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

RegisterHandle - Register property handle

The property handle is registered for being notified when an event happens on the allocated resources (index or instance). This is a precondition for receiving server events.

```
logical PropertyHandle :: RegisterHandle ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ReleaseBuffer - Release instance buffer

The function will release the allocated buffer. All instances in teh buffer are released and buffer access functions cannot be called anymore until allocating a buffer again.

```
logical PropertyHandle :: ReleaseBuffer ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

RemoveTerminator - Remove line terminator from large text fields

The function removes the terminator string (string) from the end of the text field.

```
logical PropertyHandle :: RemoveTerminator (char *string )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
string	String area Pointer to the 0-terminated string area.

Rename - Rename instance

The rename function allows changig the sort key value of the selected instance. After changig the key value the instance is stored.

The effect is the same as changing the key attributes in the instance, unless that no knowledge about the key attributes is required.

The function returns an error (YES) when no instance is selected or when

```
logical PropertyHandle :: Rename (Key new_key )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
new_key	New key for the instance The key passed for renaming the instance must be structured according to the currently selected sort order.

RepairIndex - Repair Index

The function repairs the index for the collection in the property handle. When no key name is passed, the currently selected index will be repaired. Messages about repair actions are written to the error log-file.

The function wil remove index entries pointing to invalid indanced (deleted). It repairs also index tree information in large indexes.

```
logical PropertyHandle :: RepairIndex (char *key_name_w, char
                          *attrstr_w )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
key_name_w	Key name for conversion The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks. If no key name is passed (NULL) the sort key according to the selected sort order is used instead,
attrstr_w	Attribute type The attribute type is passed as 0-terminated string. It must be one of the defined values in the enumerated value set for the attribute types defined for the generic attribute.

ReplaceSysVariables - Replace system variable

The function allows replacing system variable references in a text field. When the property handle refers to a text field that contains references to system variables (e.g. "...%SYSVAR1% ....") those references are be replaced by the text currently set for the referenced system variable (in an ini-file or by the application (-> SetSysVariable())).

When the function is called for persistent fields the updated text causes a modification and will be stored into the database. Hence, it might be better to create a copy of the property and replacing the text in the copy.

```
logical PropertyHandle :: ReplaceSysVariables ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ReplaceText - Replace system variable value

The function allows replacing text strings in a text field. When the property handle refers to a text field defined strings as passed in old_str can be replaced by the text passen in new_str.

When the function is called for persistent fields the updated text causes a modification and will be stored into the database. Hence, it might be better to create a copy of the property and replacing the text in the copy.

```
logical PropertyHandle :: ReplaceText (char *old_str, char
                           *new_str )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
old_str	Old string value
new_str	New string value

Reset - Reset instance

The function resets the current selection in the property handle, i.e. the selection is cancelled without storing the last updates (->Cancel()). Since the function is cancelling the selection all subordinated property handles will cancel the selection as well.

Than the function will re-read the instance from the database (->Get()). Subordinated property handles remain in the unselect state.

```
logical PropertyHandle :: Reset ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ResetSelection - Reset selection condition for collection

Whe a filter has been set for the property handle (-> SetSelection()) this function will reset the selection, i.e. the filter is not active anymore for the property handle. When no filter had been set for the property handle the function has no effect.

```
logical PropertyHandle :: ResetSelection ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ResetTransientProperty - Reset transient property handle

The function releases the associated property handle for a transient property handle. The associated handle will be released for the original transient property handle and all its copies. If there are no more users registered for the property handle the access node will be destroyed.

```
logical PropertyHandle :: ResetTransientProperty ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ResetWProtect - Reset permanent write protection

The function allows resetting the permanent write protection for an instance. The property handle must be opened in update or write mode and the instance must be selected.

```
logical PropertyHandle :: ResetWProtect ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Save - Store instance

The function stores all modification made on the selected instance and updates made on instances in subordinated property handles. Within a transaction the the function will write the updates to the transaction buffer. Modifications are stored to the database when the transaction is closed (Commit()).

The function is called automatically when changing the selection for a property handle and modifications have been made on the instance.

```
logical PropertyHandle :: Save (char savopt, logical switchopt )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
savopt	Store option This option indicates that instances, that have been modified meanwhile by another user, can be overwritten (YES). When passing NO for this option the function returns an error when the instance has been updated by another user after reading it into the application.
switchopt	Unselct option The option forces the function to unselect the selected instance in the property handle after terminating the function.

SearchText - Search string in property

This function performs a string search for a text property. If the property is not a text property (-> IsText()) the function returns AUTO (-1).

```
int32 PropertyHandle :: SearchText (char *string, int32 curpos,
                   logical case_opt )
```

Return value	
string	String area Pointer to the 0-terminated string area.
curpos	
case_opt	Case sensitive The option indicates case sensitive data in text (YES)

Select - Create a subset from a collection

The result collection contains the instances from the passed collection that return true (YES) for the expression passed to the select function. The expression passed must define a valid expression for the structure of the passed collection.

The operation is performed with the passed operand storing the result in the collection referenced by the calling property handle. When the calling property handle refers to a non empty collection all instances are removed before performing the operation. When the calling property handle is empty the function creates a temporary extend for storing the result.

```
PropertyHandle &PropertyHandle :: Select (PropertyHandle
                   &prophdl_ref, char *expression )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.
expression	OQL expression An OQL expression defines a condition according to the OQL syntax. OQL expressions must always terminate with ';'. The OQL-Expression is passed as 0-terminated string.

SetActionResult - Set result string

The function allows setting a result string for the property handle. The result string can be retrieved with the GetActionResult function. Thus you can pass the result of any action also to a client application while the action is running on the server. The result is passed as string, i.e. the result must not contain any 0-characters except the terminating 0.

```
void PropertyHandle :: SetActionResult (char *result_string )
```

result_string	Result string
	The result string can be a list of strings where strings are usually separated by x01 characters. If there is only one string returned the string is 0-terminated. Multiple strings are terminated with 0 after the last string in the list, which should be terminated with x01 as well.

SetArea - Set area pointer for property instance

The function can be used for property handles referring to internal data (transient fields) to allocate an instance area. The data area is not owned by the property handle in this case and will not automatically freed when closing the property handle.

Do not use this functions for subordinated property handles that refer to properties in instances. This will disconnect the property handle from its instance data.

```
Instance PropertyHandle :: SetArea (void *datarea )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area.
	You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
datarea	

SetDescription - Set definition for property

The function can be used for property handles referring to internal data (transient fields) to allocate a description (DBFieldDef). The description is not owned by the property handle in this case and will not automatically freed when closing the property handle.

Do not use this functions for subordinated property handles that refer to properties in instances.

i0

```
void PropertyHandle :: SetDescription (DBFieldDef *prop_def )
```

prop_def	Property definition The property defintion contains the metadata for the referenced property instance.

i01

```
void PropertyHandle :: SetDescription (fmcb *fmcbptr )
```

fmcbptr

SetDynLength - Activate dynamic length handling

The function activates dynamical size correction for the data area of the property handle. When assigning a value that is larger then the data area the data area will increase automatically instead of cutting the value.

```
logical PropertyHandle :: SetDynLength ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

SetGenAttribute - Set type for generic attribute in instance

The function sets the type for a generic attribute. The attribute type (as e.g. language) can be passed as string (attrstr) or type number (attrtype).

When the property handle does not refer directly to a generic attribute the property path for the generic attribute in the instance must be passed to the function (propnames).

i0

```
logical PropertyHandle :: SetGenAttribute (char *attrstr )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
attrstr	Attribute type The attribute type is passed as 0-terminated string. It must be one of the defined values in the enumerated value set for the attribute types defined for the generic attribute. When the index is not generic, no attribute needs to be passes. If no attribute is pased for a generic index the current setting is used.

i1

```
logical PropertyHandle :: SetGenAttribute (int attrtype )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
attrtype	Type of generic attribute The type for a generic attribute is a valid value from the basic enumeration of the generic attribute. UNDEF (0) indicates an undefined generic type.

i2

```
logical PropertyHandle :: SetGenAttribute (char *attrstr, char
                    *prop_path )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
attrstr	Attribute type The attribute type is passed as 0-terminated string. It must be one of the defined values in the enumerated value set for the attribute types defined for the generic attribute. When the index is not generic, no attribute needs to be passes. If no attribute is pased for a generic index the current setting is used.
prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

i3

```
logical PropertyHandle :: SetGenAttribute (int attrtype, char
                          *prop_path )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
attrtype	Type of generic attribute The type for a generic attribute is a valid value from the basic enumeration of the generic attribute. UNDEF (0) indicates an undefined generic type.
prop_path	Property path The property path is passed as 0-terminated string. It may contain a single property name or a sequence of property names separated by '.'.

SetInstance - Set basic instance for property

The function locates the area for the property handle in an instance. When defining internal structured instances this allows locating the property instance within a structure instance. This function should not be called for property handles in database instances since it may disconnect the property handle from the database instance.

```
Instance PropertyHandle :: SetInstance (char *instance )
```

Return value	Persistent instances do have the type of the referenced collection handle (collection type). Persistent instances may contain references to other instances or collections. Referenced instanced can be accessed by collection handles that are part of the persistent instance. The collection handles for referenced instances can be accessed by the property name that has been defined in the structure definition. When accessing the collection as MEMO-collection (PI(mem)) no specific instance type is provided. In this case collection handle for references can be provided via the **{.r GetPIPointer()**} function.
instance	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

SetInstanceAction - Register action in the instance context

The function adds an action to the instance context of the property handle. The instance action, which is not defined in the data model, is available in the instance context for the current property handle, only, but not for all instance contexts of this type.

```
logical PropertyHandle :: SetInstanceAction (SimpleAction
                  *action )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
action	Simple Action The simple action defines the context action and the action type. Some action types require more detailled action definitions that will be retrieved in the dictionary. In this case the dictionary must contain an appropriate action definition.

SetInstanceEventHandler - Set Instance Event Handler

Instance event handlers can be used as an alternative way to handle server notifications independent on implemented context classes. To be notified from the server the property handle must be registered on the server (RegisterPropertyHandle().

When setting event handlers in addition to a context class handler function, the context class handler is executed prior to the application handlers. When the context class handler returns an error (YES), the application handlers are not executed. Execution of application handlers is also stopped, when the first application handler returns an eror (YES).

The event handler is passed as an event link that consists of an event handler function and a class instance. The handler is called later with the instance of the event handler class set in the event link.

When adding several instance event handlers, they are called in the sequence as being added to the property handle. Handlers can be removed using the ResetInstanceEventHandler() function.

```
void PropertyHandle :: SetInstanceEventHandler (EventLink
                       *event_link )
```

event_link	Event link

An event link defines the link between a property handle and an event handler. A simple way of defining an event link is provided with the ELINK macro:

ELINK(class_instance, class_name, function_name)

SetInstanceProcessHandler - Activate Instance process event handler

Instance process event handler can be used as an alternative way to handle instance database events within an instance (structure) context class. In contrast to database event handler functions in context classes, event handler can be installed in any application without defining specific context classes. This allows handling different instances with the one event handler.

When setting event handlers in addition to a context class handler function the context class handler is executed prior to the application handlers. When the context class handler returns an error (YES), the application handlers are not executed. Execution of application handlers is also stopped, when the first application handler returns an eror (YES).

The event handler is passed as an event link that consists of an event handler function and a class instance. The handler is called later with the instance of the event handler class set in the event link.

When adding several instance process event handlers, they are called in the sequence as being added to the property handle. Handlers can be removed using the ResetInstanceProcessHandler() function.

```
void PropertyHandle :: SetInstanceProcessHandler (EventLink
                    *event_link )
```

event_link	Event link An event link defines the link between a property handle and an event handler. A simple way of defining an event link is provided with the ELINK macro: ELINK(class_instance, class_name, function_name)

SetKey - Move ident key value to instance

The passed key value is stored to the component attributes of the identifying key in the instance. When passing an empty instance the key is stored in the selected instance of the property handle. When no insztance is passed or selected or when no identifying key is defined for the structure the function returns an error (YES).

```
logical PropertyHandle :: SetKey (Key ident_key, Instance in-
                    stance_w )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
ident_key	Ident key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey() function.
instance_w	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a reference to a propertly structured area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not. Default: Instance()

SetNormalized - Set normalized value in attribute

The function can be used for storing integer values with decimal precisions in INT or unsigned INT attributes. When defining a an attribute with two decimals, assigning 1 will result inernally into 100 (1.00). Assigning the value using SetNormalized will result in 1 (0.01).

```
logical PropertyHandle :: SetNormalized (int32 long_val )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
long_val	Integer value The value is passed as platform independent 32-bit integer value.

SetOrder - Set sort order

When there are different sort orders (indexes) supported for a collection one of those can be selected as current sort order. The sort order or index is selected by the key name that is associated with the index. Changing the sort order will reset the selection in the property handle and no instance is selected when the function returns.

When selecting a generic attribute index the attribute type (as e.g. language) can be passed to select the proper index. If no attribute type is passed it is evaluated from the generic attribute.

When not passing a sort key name the default index is set as current sort order. The default index is the identifying key index (when defined for the collection) or the first unique index in the list of available indexes. Passing "*" for the sort key name refreshs the sort order. This way it is possible to set the proper index for a generic attribut index or to reorganize a tamporary index.

When no index is defined for the passed key name or the attribute type is not defined the function returns an error (YES).

i00

```
logical PropertyHandle :: SetOrder (char *key_name_w )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
key_name_w	Key name for conversion The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks. If no key name is passed (NULL) the sort key according to the selected sort order is used instead,

i01

```
logical PropertyHandle :: SetOrder (char *key_name, int attrtype
                    )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
key_name	Key name The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.
attrtype	Type of generic attribute The type for a generic attribute is a valid value from the basic enumeration of the generic attribute. UNDEF (0) indicates an undefined generic type.

i02

```
logical PropertyHandle :: SetOrder (char *key_name, char
                 *attrstr )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
key_name	Key name The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.
attrstr	Attribute type The attribute type is passed as 0-terminated string. It must be one of the defined values in the enumerated value set for the attribute types defined for the generic attribute. When the index is not generic, no attribute needs to be passes. If no attribute is pased for a generic index the current setting is used.

i03

```
logical PropertyHandle :: SetOrder (char *key_name, int at-
                 trtype, char *attrstr )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
key_name	Key name The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.
attrtype	Type of generic attribute The type for a generic attribute is a valid value from the basic enumeration of the generic attribute. UNDEF (0) indicates an undefined generic type.
attrstr	Attribute type The attribute type is passed as 0-terminated string. It must be one of the defined values in the enumerated value set for the attribute types defined for the generic attribute. When the index is not generic, no attribute needs to be passes. If no attribute is pased for a generic index the current setting is used.

SetPropertyAction - Register action in the property contect

The function adds an action to the property context of the property handle. The property action, which is not defined in the data model, is available in the instance context for the current property handle, only, but not for all instance contexts of this type.

```
logical PropertyHandle :: SetPropertyAction (SimpleAction
                 *action )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
action	Simple Action The simple action defines the context action and the action type. Some action types require more detailled action definitions that will be retrieved in the dictionary. In this case the dictionary must contain an appropriate action definition.

SetPropertyEventHandler - Set Property Event Handler

Property event handlers can be used as an alternative way to handle server notifications independent on implemented context classes. To be notified from the server the property handle must be registered on the server (RegisterPropertyHandle()).

When setting event handlers in addition to a context class handler function, the context class handler is executed prior to the application handlers. When the context class handler returns an error (YES), the application handlers are not executed. Execution of application handlers is also stopped, when the first application handler returns an eror (YES).

The event handler is passed as an event link that consists of an event handler function and a class instance. The handler is called later with the instance of the event handler class set in the event link.

When adding several property event handlers, they are called in the sequence as being added to the property handle. Handlers can be removed using the ResetPropertyEventHandler() function.

```
void PropertyHandle :: SetPropertyEventHandler (EventLink
                  *event_link )
```

event_link	Event link

An event link defines the link between a property handle and an event handler. A simple way of defining an event link is provided with the ELINK macro:

ELINK(class_instance, class_name, function_name)

SetSelection - Set filter condition for collection handle

The function allows applying a filter expression to the collection. The expression must be a valid expression in the context of the structure defined for the property handle. When a filter is set the property handle selects only those instances that return true (-> IsTrue()) for the expression. Sequential retievals as NextKey(), operators ++ and -- or Position() automatically search for the next valid instance. The Get() function that is requesting a specific instance by index or key returns an empty instance when the requested instance does not fulfill the filter condition.

When setting a filter for an update or write property handle updating an instance may lead to an invalid instance. In this case the instance is unselected after storing the updated data.

i0

```
logical PropertyHandle :: SetSelection (char *expression )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
expression	OQL expression An OQL expression defines a condition according to the OQL syntax. OQL expressions must always terminate with ';'. The OQL-Expression is passed as 0-terminated string.

i02

```
logical PropertyHandle :: SetSelection (BNFData *bdata )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

SetSortKey - Store sort key value to instance

The passed key value is stored to the component attributes of the selected sort key in the instance. When passing an empty instance the key is stored in the selected instance of the property handle. When no instance is passed or selected or when no sort key is defined for the structure the function returns an error (YES).

```
logical PropertyHandle :: SetSortKey (Key sort_key, Instance in-
                          stance_w )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
sort_key	Sort key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey}() function. When no key is passed by the application an empty key (without data area) will be passed.
instance_w	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a reference to a propertly structured area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not. Default: Instance()

SetTransientProperty - Setting property handle for transient property (reference)

This function sets the current handle for a transient property handle. The function will not create a copy of the property handle. The associated handle is registered, only.

When a copy of the referenced property handle is required the application has to create the copy befor setting it. After associating the copy for the referenced property handle it can be destroyed. Because it is registered in the transient property handle it will be destroyed when resetting the transient reference or when setting another property handle for the same transient reference.

When creating a copy of a transient reference the copy will get the same referenced handle. All referened property handles for the original transient property handle and all its copies will be updated when setting a new property handle for the original transient property handle or one of its copies.

Transient property handles are destroyed automatically when they are placed in a persistent object instance and this instance is destroyed. When, however, referring recursively to a property handle by associating a parent or higher property handle with a subordinated property handle this may result in never deleting the access node. Use MarkUsed() and MarkUnused() for handeling this situation.

You can release the associated property handle using the ResetTransientProperty() function.

```
logical PropertyHandle :: SetTransientProperty (PropertyHandle
                          &prop_hdl )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

SetType - Set type for weak-typed collection

Before creating a new instance for a weak-typed collection the type of the instance to be inserted has to be set in the property handle. The type of instance to be created is passe as structure name (strnames). This setting might be reset when reading the next instance in the collection.

```
logical PropertyHandle :: SetType (char *strnames )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
strnames	Structure name The structure name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks.

SetVersion - Set instance version to be provided by the collection handle

The function allows changing the version for a property handle. This allows providing older instance versions that are stored for the instance selected.

Passing CUR_VERSION will reset the version to the current version for the property handle.

```
logical PropertyHandle :: SetVersion (uint16 version_nr )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
version_nr	Internal version number Version numbers are created internally for each Active Object when creating a new version for the Database Object. Each version number is associated with a time stamp that defines the end of this version. Default: CUR_VERSION

SetWProtect - Set permanent write protection

The function sets permanent write protection for the selected instance. After being permanently write protected the instance cannot be updated until the write protection is reset (-> ResetWProtect()). The property handle must be opened in update or write mode and the selected instance must be writeable (-> IsWrite()).

```
logical PropertyHandle :: SetWProtect ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

StoreData - Store instance data to property handle

The passed instance is stored to the structure attributes of the selected instance. When no instance is selected or located or when the instance is not writeable the function returns an error (YES).

```
logical PropertyHandle :: StoreData (char *instance )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
instance	Instance area Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.

StringToKey - Convert string to internal key

The passed key will be converted from an extended SDF string into the internal key instance format. As field separator in the string key '|' is assumed. Structure levels are enclosed in '{}'. Normally the key passed is assumed to be structured according to the sort key selected for the property handle ot according to the identifying key (when no sort key is defined). It is, however, also possible to pass a valid key name for conversion.

```
Key PropertyHandle :: StringToKey (Key key_string, char
                       *key_name_w )
```

Return value	The key value structure corresponds to the structure of the passed or selected key.
key_string	String area for key The key is provided as ESDF key. {} are used as instance parenthesis, | is used as property delimiter. Delimiters may change when defined differently in the DataFormat option.
key_name_w	Key name for conversion The key name is passed as 0-terminated string or as buffer with a maximum size of 40 characters and trailing blanks. If no key name is passed (NULL) the sort key according to the selected sort order is used instead,

ToTop - Position to top of collection (before first)

The function positions the property handle before the first instance in the collection according to the defined sort order. Thus, a subsequent ++ operation or NextKey() will select the first instance in the collection. No instance is selected after calling this function.

When an instance is selected in the property handle it will be unselected (and stored when it was modified).

For Access pathes the function can be used to initialize the asscess path. After opening an access path the path has not been executed and is uninitialized until the next Get() call. ToTop() will inilialize the path in advance.

```
logical PropertyHandle :: ToTop ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

Union - Union two collections

The result collection contains the instancesfrom all operand collections. When passing no for the distict option the result contains also duplicates. Otherwise duplicates are not stored in the result collection. Duplicates are determined by means of sort key (passing YES for ik_opt) or local identities (LOID). Using the LOID is save but comparing the key is much faster. Hence, the key check should be used whenever possible.

i0 - Binary union

This implementation builds the union from the two collections passed to the operation. The result is stored in the collection referenced by the calling property handle. When the calling property handle refers to a non empty collection all instances are removed before performing the operation. When the calling property handle is empty the function creates a temporary extend for storing the result.

```
PropertyHandle &PropertyHandle :: Union (PropertyHandle
                    &prop_hdl1, PropertyHandle &prop_hdl2, char
                    sk_opt, logical distinct )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prop_hdl1	First Property handle Reference to an opened property handle.
prop_hdl2	Second Property handle Reference to an opened property handle.
sk_opt	Sort key option The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances. Default: YES
distinct	Distinct option Passing a distinct option YES forces the function to remove duplicates from the result collection.

i01 - Union list of collections

This implementation builds the union from all collections passed to the operation. The result is stored in the collection referenced by the calling property handle. When the calling property handle refers to a non empty collection all instances are removed before performing the operation. When the calling property handle is empty the function creates a temporary extend for storing the result.

```
PropertyHandle &PropertyHandle :: Union (PropertyHandle
                 **ph_list, int16 count, char sk_opt, logical
                 distinct )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
ph_list	List of property handles An array of property handles acting as operands in the operation. The number of property handles in the array is passed in the count-parameter.
count	Number of entries
sk_opt	Sort key option The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances. Default: YES
distinct	Distinct option Passing a distinct option YES forces the function to remove duplicates from the result collection.

i02 - Union inplace

Calling the function with one property handle creates the union collection is built from the calling and the passed collection and the result is stored in the calling collection. This will change the collection for the calliung proiperty handle.

```
PropertyHandle &PropertyHandle :: Union (PropertyHandle
                 &prophdl_ref, char sk_opt, logical distinct )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.
sk_opt	Sort key option The sortkey option indicates whether the operation should be performed according to the sort key set for the collections (YES. Otherwise (NO) the operation is performed by comparing instances. Default: YES
distinct	Distinct option Passing a distinct option YES forces the function to remove duplicates from the result collection.

Unlock - Unlock instance

This function allows unlocking the selected instance of the property handle after it has been locked (-> Lock()). Instances for shared base structures are not automatically included in the unlocking and must be unlocked separately when being locked separately.

The function returns NO when the instance has been unlocked successfully. It returns en error (YES) when the instance is not locked, when no instance is selected in the property handle or when another error occurred.

```
logical PropertyHandle :: Unlock ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

UnlockSet - Unlock collection

This function allows unlocking a collection handle referenced in a collection property handle that has been locked (-> Lock()) within the application.

The function returns NO when the collection has been unlocked successfully. It returns en error (YES) when the collection has not been locked, when no instance is selected in the upper property handle (when existing) or when another error occurred.

```
logical PropertyHandle :: UnlockSet ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

UnregisterHandle - Unregister property handle

The property handle is unregistered for being notified when an event happens on the allocated resources (index or instance).

```
logical PropertyHandle :: UnregisterHandle ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

ValidateNode - Checks whetehr the Namdle is valid

Usually a property handle is valid when it has been opened successfully. When one of the upper handles, however, is weak typed or untyped the handle may become invalid when changing the selection in the upper node.

```
logical PropertyHandle :: ValidateNode ( ) const
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

operator! - Negation operator for logical values

The negation operation performed depends on the type of the first operand. If the second operand is not compatible with the first operand the function tries to convert the second operand into the type of the first operand. If no conversion is possible the operation fails.

1. Numerical data

The operator substract the second operand from the first one.

2. Text data

The operator removes all occurences of operand 2 in operand 1, i.e. "Paul Miller" - "aul" = "P Miller".

3. Collections

The Minus operation is performed as operand1.Minus(operand2)

4. Time fields

For date and time you may substract integer or time values. values, only. Substracting an integer results in a new time value of the same type (operand1). Substracting a time value results in an integer containing the distance between the time values.

5. Logical

The substract operation returns the result of an exclusive or operation

The result is returned in a property handle that is created temporarily. You can assign the result to another property handle or performing further operations.

```
logical PropertyHandle :: operator! ( )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

operator!= - Compare two property instances (not equal)

The operation compares the two operands and returns false (NO) when they are identical and true (YES) otherwise.

i0 - Compare with other property handle

This operator compares the value in the property handle with the value in the passed property handle. Data conversion is performed when required.

```
logical PropertyHandle :: operator!= (const PropertyHandle
                  &cprop_hdl )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i01 - Compare with string value

This operator compares the value in the property handle with the value in the passed string. Data conversion is performed for the string when required.

```
logical PropertyHandle :: operator!= (char *string )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
string	String area Pointer to the 0-terminated string area.

i02 - Compare with 32-bit integer value

This operator compares the value in the property handle with the passed integer value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator!= (int32 long_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
long_val	Integer value The value is passed as platform independent 32-bit integer value.

i03 - Compare with double value

This operator compares the value in the property handle with the passed double float value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator!= (double double_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
double_val	

i04 - Compare with date value

This operator compares the value in the property handle with the passed date value. Data conversion is performed for the passed value when required. Converting date values to string values may result in different string values for the same date value depending on the national setting. Hence, string values should not be compared with date values. In this case it is more appropriate to compare the date values directly (ph.GetDate() != date_val).

```
logical PropertyHandle :: operator!= (dbdt date_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
date_val	Date value The data value is passed in the internal data format.

i05 - Compare with time value

This operator compares the value in the property handle with the passed time value. Data conversion is performed for the passed value when required. Converting time values to string values may result in different string values for the same time value depending on the national setting. Hence, string values should not be compared with time values. In this case it is more appropriate to compare the time values directly (ph.GetTime() != time_val).

```
logical PropertyHandle :: operator!= (dbtm time_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

time_val	Time value The time value is passed in the internal data format.

i06 - Compare property handles

This operator compares the property handles, i.e. the cursor objects referenced by the property handle. Property handles are considered as equal, when they refer to the same cursor.

```
logical PropertyHandle :: operator!= (PropertyHandle
                *property_handle )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
property_handle	Pointer to a property handle Is a pointer to an (usually) opened property handle.

operator% - Remaining part for integer division

The modulo operation is supported for numerical data, only. If the second operand is not compatible with the first operand the function tries to convert the second operand into the type of the first operand. If no conversion is possible the operation fails. The operator provides the remaining part of a division of the first operator by the second one.

The result is returned in a property handle that is created temporarily. You can assign the result to another property handle or performing further operations.

```
PropertyHandle PropertyHandle :: operator% (PropertyHandle
                &prop_hdl )
```

Return value	
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

operator& - AND operator (or intersect)

The intersect operation is supported for collections, only. It can be used instead of the Intersect function (Intersect(operand1,operand2)).

The result is returned in a property handle that is created temporarily. You can assign the result to another property handle or performing further operations.

```
PropertyHandle PropertyHandle :: operator& (PropertyHandle
                 &prophdl_ref )
```

Return value	
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.

operator&& - Logical AND operation

The operator returns true (YES) if both operands are true and NO otherwise.

(-> IsTrue())

```
logical PropertyHandle :: operator&& (const PropertyHandle
                 &cprop_hdl )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

operator&= - AND operator (intersect collections)

The operator returns the result of an intersec operation in the first operand.

(-> operator&)

```
PropertyHandle &PropertyHandle :: operator&= (PropertyHandle
                 &prophdl_ref )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.

operator() - Locate instance

The operator can be used to locate an instance instead of the Get() function.

(-> Get())

i0

```
Instance PropertyHandle :: operator() (int32 set_pos0 )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
set_pos0	Position in collection The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance. For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored. Special positions are **CUR_INSTANCE** CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance. **FIRST_INSTANCE** FIRST_INSATNCE (0) refers to the first instance in a collection according to the selected index (sort order). **LAST_INSTANCE** FIRST_INSATNCE (0) refers to the last instance in a collection according to the selected index (sort order).

i02

```
Instance PropertyHandle :: operator() (PropertyHandle &prop_hdl
                 )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i1

```
Instance PropertyHandle :: operator() (Key sort_key )
```

Return value	Instances do have the type of the referenced property handle (collection type). The instance contains a pointer to a properly structured instance area. You can use the !-operator or the GetData() function to check whether the instance refers to data or not.
sort_key	Sort key value The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the StringToKey}() function. When no key is passed by the application an empty key (without data area) will be passed.

operator* - Multiply two properties

The multiplication operation is supported for some types, only. The way the operation is performed depends on the type of the first operand. If the second operand is not compatible with the first operand the function tries to convert the second operand into the type of the first operand. If no conversion is possible the operation fails.

1. Numerical data

The operator multiplies the second operand with the first one.

5. Logical

The multiplication returns the result of an and operation

The result is returned in a property handle that is created temporarily. You can assign the result to another property handle or performing further operations.

```
PropertyHandle PropertyHandle :: operator* (const PropertyHandle
                 &cprop_hdl )
```

Return value	
cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

operator*= - Multiply and assign result to first operator

The operator returns the result of a multiplication in the first operand.

(-> operator*)

```
PropertyHandle &PropertyHandle :: operator*= (const PropertyHan-
                 dle &cprop_hdl )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

operator+ - Sum two properties

The sum operation performed depends on the type of the first operand. If the second operand is not compatible with the first operand the function tries to convert the second operand into the type of the first operand. If no conversion is possible the operation fails.

1. Numerical data

The operator adds the second operand to the first one.

2. Text data

The operator concatenates the second operand to the first operand 1, i.e. "Paul " + "Miller" = "Paul Miller".

3. Collections

The Union operation is performed as operand1.Union(operand2)

4. Time fields

For date and time you may add only integer.

5. Logical

The substract operation returns the result of an or operation

The result is returned in a property handle that is created temporarily. You can assign the result to another property handle or performing further operations.

```
PropertyHandle PropertyHandle :: operator+ (const PropertyHandle
                  &cprop_hdl )
```

Return value	
cprop_hdl	Property Handle
	Is a reference to an (usually) opened property handle.

operator++ - Position cursor on next instance

The increment operation performed depends on the type of the operand.

1. Numerical data

The operator increments the value by 1.

2. Collections

The operation tries to locate the next instance in the collection. If no iinstance is selected it locates the first instance n the collection.

The result is returned in the operand.

i0

```
PropertyHandle &PropertyHandle :: operator++ (int  )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).

i01

```
PropertyHandle &PropertyHandle :: operator++ ( )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).

operator+= - Sum and assign result to first operator

The operator returns the sum of the two operands in the first operand.

(-> operator+)

```
PropertyHandle &PropertyHandle :: operator+= (const PropertyHan-
                dle &cprop_hdl )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

operator- - Subtract properties

The substract operation performed depends on the type of the first operand. If the second operand is not compatible with the first operand the function tries to convert the second operand into the type of the first operand. If no conversion is possible the operation fails.

1. Numerical data

The operator substract the second operand from the first one.

2. Text data

The operator removes all occurences of operand 2 in operand 1, i.e. "Paul Miller" - "aul" = "P Miller".

3. Collections

The Minus operation is performed as operand1.Minus(operand2)

4. Time fields

For date and time you may substract integer or time values. values, only. Substracting an integer results in a new time value of the same type (operand1). Substracting a time value results in an integer containing the distance between the time balues.

5. Logical

The substract operation returns the result of an exclusive or operation

The result is returned in a property handle that is created temporarily. You can assign the result to another property handle or performing further operations.

i0

```
PropertyHandle PropertyHandle :: operator- (const PropertyHandle
                  &cprop_hdl )
```

Return value

cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i1

```
PropertyHandle PropertyHandle :: operator- ( )
```

Return value	

operator-- - Position cursor on previous instance

The decrement operation performed depends on the type of the operand.

1. Numerical data

The operator decrements the value by 1.

2. Collections

The operation tries to locate the prevoius instance in the collection. If no iinstance is selected it locates the last instance n the collection.

The result is returned in the operand.

i0

```
PropertyHandle &PropertyHandle :: operator-- ( )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).

i01

```
PropertyHandle &PropertyHandle :: operator-- (int  )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).

operator-= - Subtract and assign result to first operator

The operator returns the difference of the first and the second operator in the first operand.

(-> operator-)

```
PropertyHandle &PropertyHandle :: operator-= (const PropertyHan-
                               dle &cprop_hdl )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
cprop_hdl	Property Handle

Is a reference to an (usually) opened property handle.

operator/ - Devide proprties

The division operation is supported for numerical data, only. If the second operand is not compatible with the first operand the function tries to convert the second operand into the type of the first operand. If no conversion is possible the operation fails. The operator devides the first operand by the second one.

The result is returned in a property handle that is created temporarily. You can assign the result to another property handle or performing further operations.

```
PropertyHandle PropertyHandle :: operator/ (const PropertyHandle
                 &cprop_hdl )
```

Return value	
cprop_hdl	Property Handle
	Is a reference to an (usually) opened property handle.

operator/= - Divide and assign result to first operator

The operator returns the result of a division in the first operand.

(-> operator/)

```
PropertyHandle &PropertyHandle :: operator/= (const PropertyHan-
                 dle &cprop_hdl )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
cprop_hdl	Property Handle
	Is a reference to an (usually) opened property handle.

operator< - Compare two property instances (less)

The operation compares the two operands and returns true (YES) when the first operand is smaller than the second operand and false (NO) otherwise.

i0 - Compare with other property handle

This operator compares the value in the property handle with the value in the passed property handle. Data conversion is performed when required.

```
logical PropertyHandle :: operator< (PropertyHandle &prop_hdl )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i01 - Compare with string value

This operator compares the value in the property handle with the value in the passed string. Data conversion is performed for the string when required.

```
logical PropertyHandle :: operator< (char *string )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
string	String area Pointer to the 0-terminated string area.

i02 - Compare with 32-bit integer value

This operator compares the value in the property handle with the passed integer value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator< (int32 long_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
long_val	Integer value The value is passed as platform independent 32-bit integer value.

i03 - Compare with double value

This operator compares the value in the property handle with the passed double float value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator< (double double_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
double_val	

i04 - Compare with date value

This operator compares the value in the property handle with the passed date value. Data conversion is performed for the passed value when required. Converting date values to string values may result in different string values for the same date value depending on the national setting. Hence, string values should not be compared with date values. In this case it is more appropriate to compare the date values directly (ph.GetDate() < date_val)

```
logical PropertyHandle :: operator< (dbdt date_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
date_val	Date value The data value is passed in the internal data format.

i05 - Compare with time value

This operator compares the value in the property handle with the passed time value. Data conversion is performed for the passed value when required. Converting time values to string values may result in different string values for the same time value depending on the national setting. Hence, string values should not be compared with time values. In this case it is more appropriate to compare the time values directly (ph.GetTime() < time_val).

```
logical PropertyHandle :: operator< (dbtm time_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
time_val	Time value The time value is passed in the internal data format.

operator<= - Compare two property instances (less or equal)

The operation compares the two operands and returns true (YES) when the first operand is smaller than or equal to the second operand and false (NO) otherwise.

i0 - Compare with other property handle

This operator compares the value in the property handle with the value in the passed property handle. Data conversion is performed when required.

```
logical PropertyHandle :: operator<= (PropertyHandle &prop_hdl )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i01 - Compare with string value

This operator compares the value in the property handle with the value in the passed string. Data conversion is performed for the string when required.

```
logical PropertyHandle :: operator<= (char *string )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
string	String area Pointer to the 0-terminated string area.

i02 - Compare with 32-bit integer value

This operator compares the value in the property handle with the passed integer value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator<= (int32 long_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
long_val	Integer value The value is passed as platform independent 32-bit integer value.

i03 - Compare with time value

This operator compares the value in the property handle with the passed time value. Data conversion is performed for the passed value when required. Converting time values to string values may result in different string values for the same time value depending on the national setting. Hence, string values should not be compared with time values. In this case it is more appropriate to compare the time values directly (ph.GetTime() <= time_val).

```
logical PropertyHandle :: operator<= (dbtm time_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
time_val	Time value The time value is passed in the internal data format.

i04 - Compare with date value

This operator compares the value in the property handle with the passed date value. Data conversion is performed for the passed value when required. Converting date values to string values may result in different string values for the same date value depending on the national setting. Hence, string values should not be compared with date values. In this case it is more appropriate to compare the date values directly (ph.GetDate() <= date_val).

```
logical PropertyHandle :: operator<= (dbdt date_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
date_val	Date value The data value is passed in the internal data format.

i05 - Compare with double value

This operator compares the value in the property handle with the passed double float value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator<= (double double_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
double_val	

operator= - Assign property instances

The assignment operator allows assigning values of property handles to each other. The way the operation is performed depends mainly on the first operand:

1. First operand is (not opened)

The function copies the handle pointer from the second operand to the first operand.

(-> CopyHandle(property_handle))

2. Collection

When both, the first and the second operator, are collections, the instances in the first collection are deleted and the instances from the second collection are copied to the first collection.

3. Instance or value

if the first operand refers to an instance or value the function converts the instance or value from the second operand into the instance of the first operand. If the second operand is a collection the selected instance in this collection is copied. If no instance is selected in the second operand the function tries to select an instance in the second operand for performing the operation. Copying instances is done by assigning all properties with the same name. Copying values will perform automatic data conversion when necessary.

The operand allows also assigning values to a property handle. In this case (second operand is a value and not a property handle) the first operand must refer to a value or instance. The value is converted when necessary.

i0

```
PropertyHandle &PropertyHandle :: operator= (const PropertyHan-
                        dle &cprop_hdl )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).

cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i1

```
PropertyHandle &PropertyHandle :: operator= (int32 long_val )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
long_val	Integer value The value is passed as platform independent 32-bit integer value.

i2

```
PropertyHandle &PropertyHandle :: operator= (double dbl_value )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
dbl_value	Double value

i3

```
PropertyHandle &PropertyHandle :: operator= (dbdt date_val )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
date_val	Date value The data value is passed in the internal data format.

i4

```
PropertyHandle &PropertyHandle :: operator= (dbtm time_val )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
time_val	Time value The time value is passed in the internal data format.

i5

```
PropertyHandle &PropertyHandle :: operator= (int16 short_val )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
short_val	

i6

```
PropertyHandle &PropertyHandle :: operator= (char *string )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
string	String area Pointer to the 0-terminated string area.

i7

```
PropertyHandle &PropertyHandle :: operator= (logical logval )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
logval	Logical value Is a logical (bool) value.

i8

```
PropertyHandle &PropertyHandle :: operator= (dttm datetime_val )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
datetime_val	Date-Time value A date-time value or time point is passed in the internal date-time format.

operator== - Compare two property instances (equal)

The operation compares the two operands and returns true (YES) when the first operand is equal to then the second operand and false (NO) otherwise.

i0 - Compare with other property handle

This operator compares the value in the property handle with the value in the passed property handle. Data conversion is performed when required.

```
logical PropertyHandle :: operator== (const PropertyHandle
                     &cprop_hdl )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i01 - Compare with string value

This operator compares the value in the property handle with the value in the passed string. Data conversion is performed for the string when required.

```
logical PropertyHandle :: operator== (char *string )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
string	String area Pointer to the 0-terminated string area.

i02 - Compare with 32-bit integer value

This operator compares the value in the property handle with the passed integer value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator== (int32 long_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
long_val	Integer value The value is passed as platform independent 32-bit integer value.

i03 - Compare with double value

This operator compares the value in the property handle with the passed double float value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator== (double double_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
double_val	

i04 - Compare with date value

This operator compares the value in the property handle with the passed date value. Data conversion is performed for the passed value when required. Converting date values to string values may result in different string values for the same date value depending on the national setting. Hence, string values should not be compared with date values. In this case it is more appropriate to compare the date values directly (ph.GetDate() == date_val).

```
logical PropertyHandle :: operator== (dbdt date_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
date_val	Date value The data value is passed in the internal data format.

i05 - Compare with time value

This operator compares the value in the property handle with the passed time value. Data conversion is performed for the passed value when required. Converting time values to string values may result in different string values for the same time value depending on the national setting. Hence, string values should not be compared with time values. In this case it is more appropriate to compare the time values directly (ph.GetTime() == time_val).

```
logical PropertyHandle :: operator== (dbtm time_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.

time_val	Time value The time value is passed in the internal data format.

i06 - Compare property handles

This operator compares the property handles, i.e. the cursor objects referenced by the property handle. Property handles are considered as equal, when they refer to the same cursor.

```
logical PropertyHandle :: operator== (PropertyHandle
                         *property_handle )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
property_handle	Pointer to a property handle Is a pointer to an (usually) opened property handle.

operator> - Compare two property instances (greater)

The operation compares the two operands and returns true (YES) when the first operand is greater than the second operand and false (NO) otherwise.

i0 - Compare with other property handle

This operator compares the value in the property handle with the value in the passed property handle. Data conversion is performed when required.

```
logical PropertyHandle :: operator> (PropertyHandle &prop_hdl )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i01 - Compare with string value

This operator compares the value in the property handle with the value in the passed string. Data conversion is performed for the string when required.

```
logical PropertyHandle :: operator> (char *string )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
string	String area Pointer to the 0-terminated string area.

i02 - Compare with 32-bit integer value

This operator compares the value in the property handle with the passed integer value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator> (int32 long_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
long_val	Integer value The value is passed as platform independent 32-bit integer value.

i03 - Compare with double value

This operator compares the value in the property handle with the passed double float value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator> (double double_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
double_val	

i04 - Compare with date value

This operator compares the value in the property handle with the passed date value. Data conversion is performed for the passed value when required. Converting date values to string values may result in different string values for the same date value depending on the national setting. Hence, string values should not be compared with date values. In this case it is more appropriate to compare the date values directly (ph.GetDate() > date_val).

```
logical PropertyHandle :: operator> (dbdt date_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
date_val	Date value The data value is passed in the internal data format.

i05 - Compare with time value

This operator compares the value in the property handle with the passed time value. Data conversion is performed for the passed value when required. Converting time values to string values may result in different string values for the same time value depending on the national setting. Hence, string values should not be compared with time values. In this case it is more appropriate to compare the time values directly (ph.GetTime() > time_val).

```
logical PropertyHandle :: operator> (dbtm time_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
time_val	Time value The time value is passed in the internal data format.

operator>= - Compare two property instances (greater or equal)

The operation compares the two operands and returns true (YES) when the first operand is equal to or greater than the second operand and false (NO) otherwise.

i0 - Compare with other property handle

This operator compares the value in the property handle with the value in the passed property handle. Data conversion is performed when required.

```
logical PropertyHandle :: operator>= (PropertyHandle &prop_hdl )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i01 - Compare with string value

This operator compares the value in the property handle with the value in the passed string. Data conversion is performed for the string when required.

```
logical PropertyHandle :: operator>= (char *string )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
string	String area Pointer to the 0-terminated string area.

i02 - Compare with 32-bit integer value

This operator compares the value in the property handle with the passed integer value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator>= (int32 long_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
long_val	Integer value The value is passed as platform independent 32-bit integer value.

i03 - Compare with double value

This operator compares the value in the property handle with the passed double float value. Data conversion is performed for the passed value when required.

```
logical PropertyHandle :: operator>= (double double_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
double_val	

i04 - Compare with time value

This operator compares the value in the property handle with the passed time value. Data conversion is performed for the passed value when required. Converting time values to string values may result in different string values for the same time value depending on the national setting. Hence, string values should not be compared with time values. In this case it is more appropriate to compare the time values directly (ph.GetTime() >= time_val).

```
logical PropertyHandle :: operator>= (dbtm time_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
time_val	Time value The time value is passed in the internal data format.

i05 - Compare with date value

This operator compares the value in the property handle with the passed date value. Data conversion is performed for the passed value when required. Converting date values to string values may result in different string values for the same date value depending on the national setting. Hence, string values should not be compared with date values. In this case it is more appropriate to compare the date values directly (ph.GetDate() >= date_val).

```
logical PropertyHandle :: operator>= (dbdt date_val )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
date_val	Date value The data value is passed in the internal data format.

operator[] - Locate property instance

The operator can be used to provide an instance instead of the Provide() function.

(-> Provide())

i0

```
PropertyHandle &PropertyHandle :: operator[] (int32 set_pos0 )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
set_pos0	Position in collection The position of an instance in a collection depends on the selected index. if the collection is unsorted the position is the only way for accessing the instance. For sorted collections the position is determined according to the instance key. If thee is a contradiction between position and key value the position will be ignored. Special positions are **CUR_INSTANCE** CUR_INSTANCE refers to the currently selected instance. If no instance is selected it refers to the first instance. **FIRST_INSTANCE** FIRST_INSATNCE (0) refers to the first instance in a collection according to the selected index (sort order). **LAST_INSTANCE** FIRST_INSATNCE (0) refers to the last instance in a collection according to the selected index (sort order).

i02

```
PropertyHandle &PropertyHandle :: operator[] (PropertyHandle
                     &prop_hdl )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prop_hdl	Property Handle Is a reference to an (usually) opened property handle.

i1

```
PropertyHandle &PropertyHandle :: operator[] (void *skey )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).

skey	Sort key The key is provided in the internal key format. When necessary the key value can be converted from a string into the internal format using the ({.r pib.StringToKey}()) function. Regardles on the type key values are passed as (char *) areas.

operator^ - Exclusive OR operation

The operator returns true (YES) if exactly one of the operands is true and NO otherwise.

(-> IsTrue())

```
logical PropertyHandle :: operator^ (const PropertyHandle
                 &cprop_hdl ) const
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

operator| - OR operation (union set for collections)

The union operation is supported for collections, only. It can be used instead of the Union() function (Union(operand1,operand2)).

The result is returned in a property handle that is created temporarily. You can assign the result to another property handle or performing further operations.

```
PropertyHandle PropertyHandle :: operator| (PropertyHandle
                 &prophdl_ref )
```

Return value	
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.

operator|= - OR operation (union set for collections)

The operator returns the result of a union operation in the first operand.

(-> operator|)

```
PropertyHandle &PropertyHandle :: operator|= (PropertyHandle
                 &prophdl_ref )
```

Return value	Reference to the property handle that contains the result of an operation (usually the calling property handle).
prophdl_ref	Reference to Property handle Is a reference to an (usually) opened property handle.

operator|| - Logical OR operation

The operator returns true (YES) if one of the operands is true and NO otherwise.

(-> IsTrue())

```
logical PropertyHandle :: operator|| (const PropertyHandle
                     &cprop_hdl )
```

Return value	The function returns YES when the question was answered positivly. Otherwise it returns NO.
cprop_hdl	Property Handle Is a reference to an (usually) opened property handle.

~PropertyHandle - Destructor

The function wil close the property handle and destroy the handle object.

```
                             PropertyHandle :: ~PropertyHandle ( )
```

UtilityHandle -

CloseDAT -

```
logical UtilityHandle :: CloseDAT ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

CloseDataSource1 -

```
logical UtilityHandle :: CloseDataSource1 ( )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.

NeverCalled -

```
void UtilityHandle :: NeverCalled ( )
```

OpenDAT -

```
DataSourceHandle *UtilityHandle :: OpenDAT (PIACC accopt, logi-
                cal netopt, logical sysappl )
```

Return value	The data source handle contains definitions for external and internal resources (resource names and opened resource handles)
accopt	Access option The access option defines the way instances in a property handle are to be accessed (read, update, write).
netopt	

OpenDataSource1 -

```
logical UtilityHandle :: OpenDataSource1 (char *dbname )
```

Return value	The value is YES if the function returns an error. In case of normal termination the value is NO. When the function returns YES more detailed error information are available in the error object.
dbname	

OpenRES -

```
DataSourceHandle *UtilityHandle :: OpenRES ( )
```

Return value	The data source handle contains definitions for external and internal resources (resource names and opened resource handles)

OpenSYS -

```
DataSourceHandle *UtilityHandle :: OpenSYS ( )
```

Return value	The data source handle contains definitions for external and internal resources (resource names and opened resource handles)

~UtilityHandle -

```
UtilityHandle :: ~UtilityHandle ( )
```