

SIEMENS



HiPath 4000 ComWin 300

Programmers Guide / Reference

Guide



1P P30357-B9800-B10-101 EN

The information in this document contains only general descriptions or general features, which may not be available as described for actual application and/or which can change due to further product development.

The desired features are therefore only binding if formally specified at the closing of the contract.



The device conforms to the EU directive 1999/5/EG, as attested by the CE mark.



This device has been manufactured in accordance with our certified environmental management system (ISO 14001). This process ensures that energy consumption and the use of primary raw materials are kept to a minimum, thus reducing waste production.

Content

1 ComWin 300 Architecture	5
2 The CBL Language	7
2.1 Syntax description	7
2.1.1 program	7
2.1.2 statement	8
2.1.3 assign_stmt	9
2.1.4 call_stmt	10
2.1.5 syn_stmt	10
2.1.6 enum_stmt	11
2.1.7 set_stmt	12
2.1.8 text_table_stmt	12
2.1.9 signature_stmt	13
2.1.10 empty_stmt	13
2.1.11 include_stmt	13
2.1.12 import_stmt	14
2.1.13 function_stmt	15
2.1.14 function_parameters	15
2.1.15 declaration_stmt	17
2.1.16 class_stmt	21
2.1.17 class_member	22
2.1.18 ole/dll_cls_member	24
2.1.19 ole_type	24
2.1.20 if_stmt	25
2.1.21 for_stmt	27
2.1.22 while_stmt	28
2.1.23 do_while_stmt	29
2.1.24 try_catch_stmt	30
2.1.25 throw_stmt	30
2.1.26 with_stmt	31
2.1.27 return_stmt	32
2.1.28 break_stmt	33
2.1.29 block	33
2.1.30 expression	35
2.1.31 binop	36
2.1.32 type_identifier	41
2.1.33 identifier	42
2.1.34 alpha	43
2.1.35 digit	43
2.1.36 number	43
2.1.37 hex	43

2.1.38	quotstring	44
2.1.39	comment (structured)	45
2.1.40	comment (single line)	45
2.2	Summary of keywords.	46
2.3	Advanced topics	47
2.3.1	Working with arrays	47
2.3.2	Working with reference variables	48
2.3.3	Using the OLE interface	53
2.3.4	Using the DLL interface	56
3	CBL - Programmer's Reference	58
3.1	Interface to the Interpreter.	58
3.1.1	Command line parameters: Args	58
3.1.2	Interpreter functions	59
3.2	General functions	63
3.2.1	cbl_sysfunc_ (ole_class)	63
3.2.2	cbl_chain (ole_class)	75
3.2.3	cblFile (ole_class)	76
3.2.4	cblIniFile (ole_class)	78
3.2.5	cblZipFile (ole_class)	79
3.2.6	cbl_dialogs (ole_class)	79
3.2.7	cblXmlFile (ole_class)	80
3.3	Interfaces to the ComWin-Frame components	82
3.3.1	Comedit (ole_class)	82
3.3.2	ComwinControl (ole_class)	98
3.3.3	ComwinFrame (ole_class)	109
3.3.4	ComwinCBL (ole_class)	113
3.3.5	ComWinFT (ole_class)	114
3.3.6	ComWinPPP (ole_class)	120
3.3.7	ComWinAccessServer (ole_class)	122
3.3.8	ComWinAccessSession (ole_class)	122
3.3.9	ComWinAccessFileman (ole_class)	123
3.3.10	ComWinAccessDatabase (ole_class)	124
3.3.11	ComWinAccessQuery (ole_class)	125
3.3.12	ComWinFM (ole_class)	128
3.4	Designing forms	130
3.4.1	Form Layout	130
3.4.2	The form class cblForm	131
3.4.3	Form controls	136
3.4.4	Definition of menus.	147
3.4.5	Form examples.	148
3.4.6	Modal dialog example.	152
3.5	CBL - National Language Support	153
3.5.1	CBL-intrinsic text tables	153
3.5.2	NLS texts from H.O.T.	154

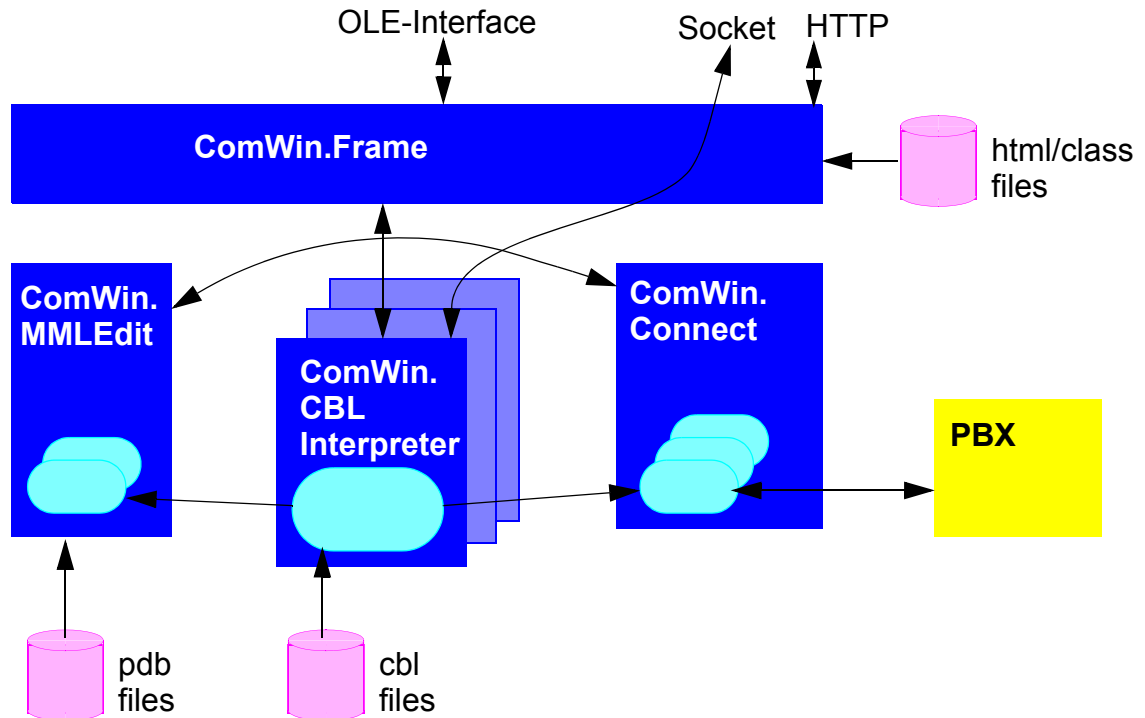
4 Introduction into CBL Programming	155
4.1 Preface	155
4.1.1 What is CBL?	155
4.1.2 What do we need to start?	156
4.2 Data types, variables and operators	158
4.2.1 Primitive data types	158
4.2.2 Operators	167
4.3 Statement and Expression	175
4.3.1 Expression	176
4.3.2 Statement flow control	176
4.4 Object and Class	182
4.4.1 What is an object?	182
4.4.2 Object instantiation	183
4.4.3 Class	184
4.5 Programming with GUI-Builder	188
4.6 ComWin Interoperation	193
1 HiPath 4000 Expert Access Architecture	7
2 Latest Enhancements	9
2.1 Integration of the SOAP Interface	9
2.2 Import of XML Schema Files (XSD)	9
2.3 Import of ASN.1 specifications	9
3 The CBL Language	10
3.1 Syntax description	10
3.1.1 program	10
3.1.2 statement	11
3.1.3 assign_stmt	13
3.1.4 call_stmt	14
3.1.5 syn_stmt	14
3.1.6 enum_stmt	15
3.1.7 set_stmt	16
3.1.8 text_table_stmt	16
3.1.9 signature_stmt	17
3.1.10 empty_stmt	17
3.1.11 include_stmt	17
3.1.12 import_stmt	18
3.1.13 function_stmt	19
3.1.14 function_parameters	20
3.1.15 declaration_stmt	22
3.1.16 class_stmt	25
3.1.17 class_member	26
3.1.18 ole/dll_cls_member	28
3.1.19 ole_type	28
3.1.20 if_stmt	29

3.1.21	for_stmt.	31
3.1.22	while_stmt.	32
3.1.23	do_while_stmt.	33
3.1.24	try_catch_stmt	34
3.1.25	throw_stmt	34
3.1.26	with_stmt	35
3.1.27	return_stmt	36
3.1.28	break_stmt	37
3.1.29	.try_catch_stmt.	38
3.1.30	block.	38
3.1.31	expression	39
3.1.32	binop.	40
3.1.33	type_identifier	45
3.1.34	identifier	46
3.1.35	alpha.	47
3.1.36	digit.	47
3.1.37	number	47
3.1.38	hex	47
3.1.39	quotstring	48
3.1.40	comment (structured)	49
3.1.41	comment (single line)	49
3.2	Summary of keywords.	50
3.3	Advanced topics	51
3.3.1	Working with arrays	51
3.3.2	Working with reference variables	53
3.3.3	Using the OLE interface	57
3.3.4	Using the DLL interface	60
3.3.5	Working with the SOAP Interface.	62
3.3.6	Import of XSD Files	64
3.3.7	Import of ASN.1-Modules	67
4	CBL - Programmer's Reference	71
4.1	Interface to the Interpreter.	71
4.1.1	Command line parameters: Args	71
4.1.2	Interpreter functions	72
4.2	General functions	76
4.2.1	cbl_sysfunc_ (ole_class)	76
4.2.2	cbl_chain (ole_class)	89
4.2.3	cblFile (ole_class).	90
4.2.4	cblIniFile (ole_class).	92
4.2.5	cblZipFile (ole_class)	93
4.2.6	cbl_dialogs (ole_class).	93
4.2.7	cblXmlFile (ole_class).	94
4.3	Interfaces to the HiPath 4000 Expert AccessFrame components.	96
4.3.1	Comedit (ole_class)	96

4.3.2 ComwinControl (ole_class)	114
4.3.3 (mode =2: application handles wrong password, open returns immediately; mode =4: deactivate autologon.)ComwinFrame (ole_class)	125
4.3.4 ComwinCBL (ole_class)	129
4.3.5 ComWinFT (ole_class)	130
4.3.6 ComWinPPP (ole_class)	136
4.3.7 ComWinAccessServer (ole_class)	138
4.3.8 ComWinAccessSession (ole_class)	138
4.3.9 ComWinAccessFileman (ole_class)	139
4.3.10 ComWinAccessDatabase (ole_class)	140
4.3.11 ComWinAccessQuery (ole_class)	141
4.3.12 ComWinFM (ole_class)	144
4.4 Designing forms	146
4.4.1 Form Layout	146
4.4.2 The form class cblForm	148
4.4.3 Form controls	152
4.4.4 Definition of menus	168
4.4.5 Form examples	169
4.4.6 Modal dialog example	173
4.5 CBL - National Language Support	174
4.5.1 CBL-intrinsic text tables	174
4.5.2 NLS texts from H.O.T.	175
5 Introduction into CBL Programming	176
5.1 Preface	176
5.1.1 What is CBL?	176
5.1.2 What do we need to start?	177
5.2 Data types, variables and operators	179
5.2.1 Primitive data types	179
5.2.2 Operators	188
5.3 Statement and Expression	196
5.3.1 Expression	197
5.3.2 Statement flow control	197
5.4 Object and Class	203
5.4.1 What is an object?	203
5.4.2 Object instantiation	204
5.4.3 Class	205
5.5 Programming with GUI-Builder	209
5.6 HiPath 4000 Expert Access Interoperation	214

1 HiPath 4000 Expert Access Architecture

This chapter gives a brief overview of the HiPath 4000 Expert Access architecture.



The HiPath 4000 Expert Access architecture is characterized by several program components which communicate via the OLE automation interface. This interface makes it possible to execute a procedure or access a property contained in another program. The other program must offer this procedure or property.

The main components of the HiPath 4000 Expert Access Framework are:

- HiPath 4000 Expert Access - CBL Interpreter: An interpreter and debugger for the new programming language "CBL" as an alternative to the MACRO language of PC-Comtes.
- HiPath 4000 Expert Access - Connect: The program controlling and keeping connections to one or more switches. Complete (and of course enhanced) functionality of PC-Comtes
- HiPath 4000 Expert Access - MML Editor: The MML command editor. ("Comedit-Window of PC-Comtes")

HiPath 4000 Expert Access Architecture

- HiPath 4000 Expert Access - Frame: A frame application which can be seen as HiPath 4000 Expert Access-entry point for applications which want use HiPath 4000 Expert Access as Server (via OLE or HTTP-interface).
- HiPath 4000 Expert Access - FileTransfer: An OLE server application with which files can be transported using various protocols.
- HiPath 4000 Expert Access - GuiBuilder: The prototype of a tool to support the creation of CBL programs, see *"Designing forms" on page 146*.

The CBL intrinsic OLE interface can be used to contact 3rd Party or owner written programs as well (e. g. Microsoft Word, Excel, Netscape Navigator).

The HTTP-Server which is part of the CBL frame is not in the focus of this document. It offers the opportunity to communicate between Java applets and CBL programs.

Focus for this document is the description of the CBL language, the description of the interfaces and the usage of the CBL Interpreter/Debugger.

2 Latest Enhancements

2.1 Integration of the SOAP Interface

By means of an enhancement of the import statement (see *"import_stmt" on page 18*) WSDL files (WebServiceDescriptionLanguage) can be imported directly into CBL programs. This can be done either as file or via HTTP. The methods that are implemented in the WebService can be called like "ordinary" CBL functions. Please refer to chapter 3.3.5, "Working with the SOAP Interface" for more information.

2.2 Import of XML Schema Files (XSD)

Due to another enhancement of the import statement (see *"import_stmt" on page 18*) XSD files (XML Schema Definitions) can be imported directly into CBL programs. All contained data structure definitions get corresponding CBL data types. Additionally each generated class gets the methods `ToString` and `FromString`. These functions can be used to serialize the current object into a XML string and vice versa. By this means a validating XML parser got integrated into CBL.

See chapter 3.3.6, "Import of XSD Files" for details.

The most complex XSD files that have been imported till now were the ECMA standardized CSTA-XML schema files.

2.3 Import of ASN.1 specifications

One further enhancement of the import statement (see *"import_stmt" on page 18*) allows to import ASN.1 modules into CBL. In this case an enhanced version of the freeware tool "snacc" is called in background to generate temporary CBL modules. These modules contain the BER decoders and encoders. They are linked to the current CBL project automatically.

The different ASN.1 variants and its functionality limit the possibilities of that feature. However, ASN.1 specifications for ACSE, CSTA and ACL-C+ (HiPath 4000) have been imported successfully.

Please refer to chapter 3.3.7, "Import of ASN.1-Modules" for details.

3 The CBL Language

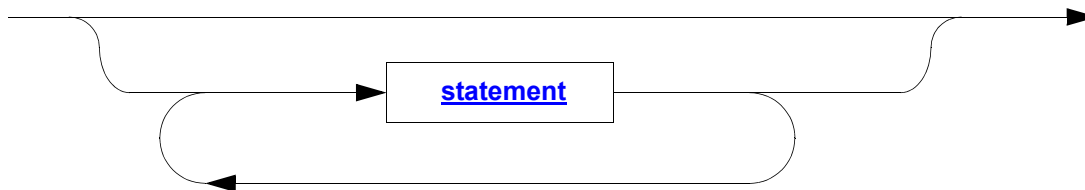
- > chapter 3.1, "Syntax description"
- > chapter 3.2, "Summary of keywords"
- > chapter 3.3, "Advanced topics"

3.1 Syntax description

This chapter describes the syntax of the CBL language.

See also chapter 3.2, "Summary of keywords".

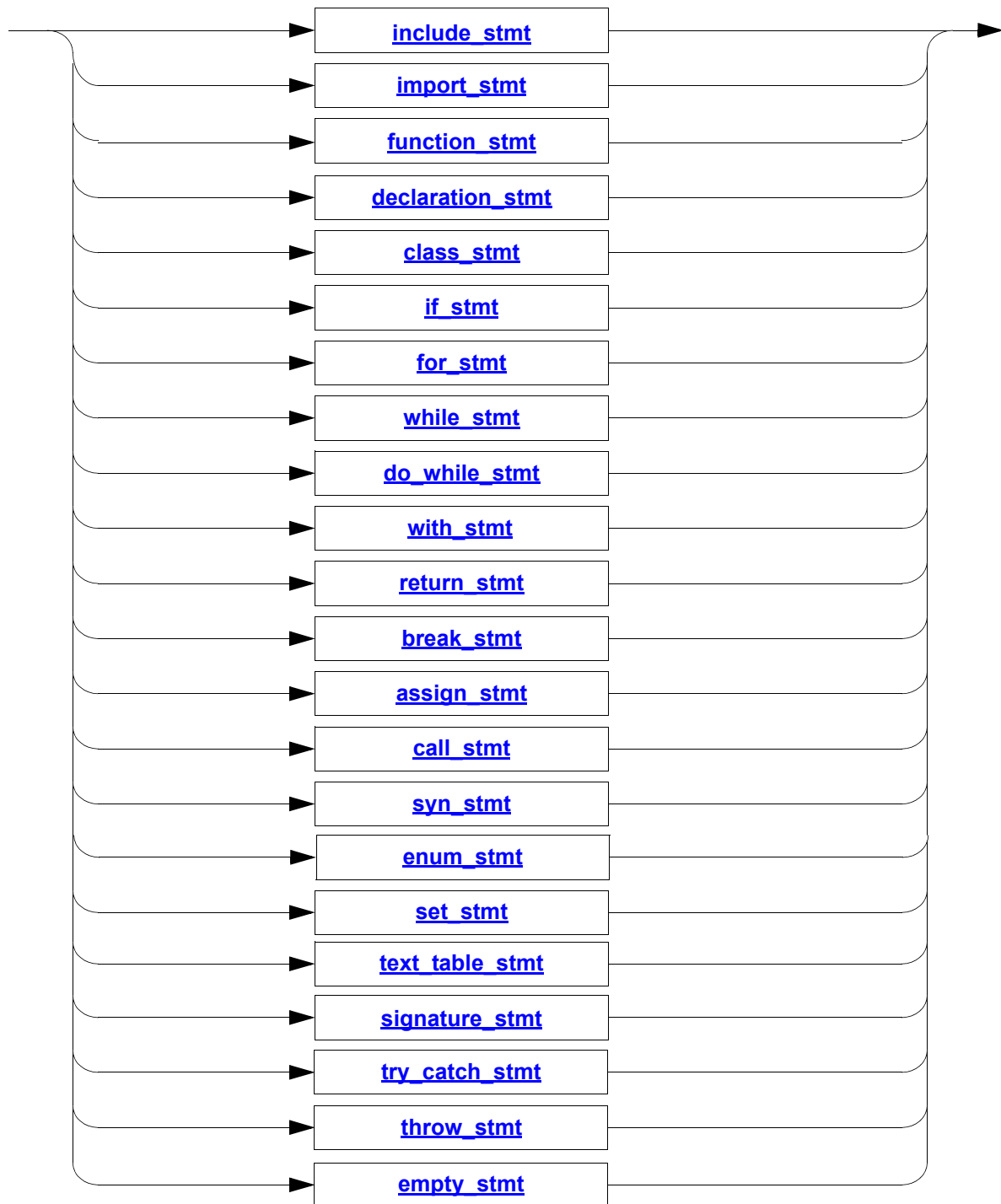
3.1.1 program



A CBL program consists of one or more statements. However, there is also the possibility that the program is empty without any statement.

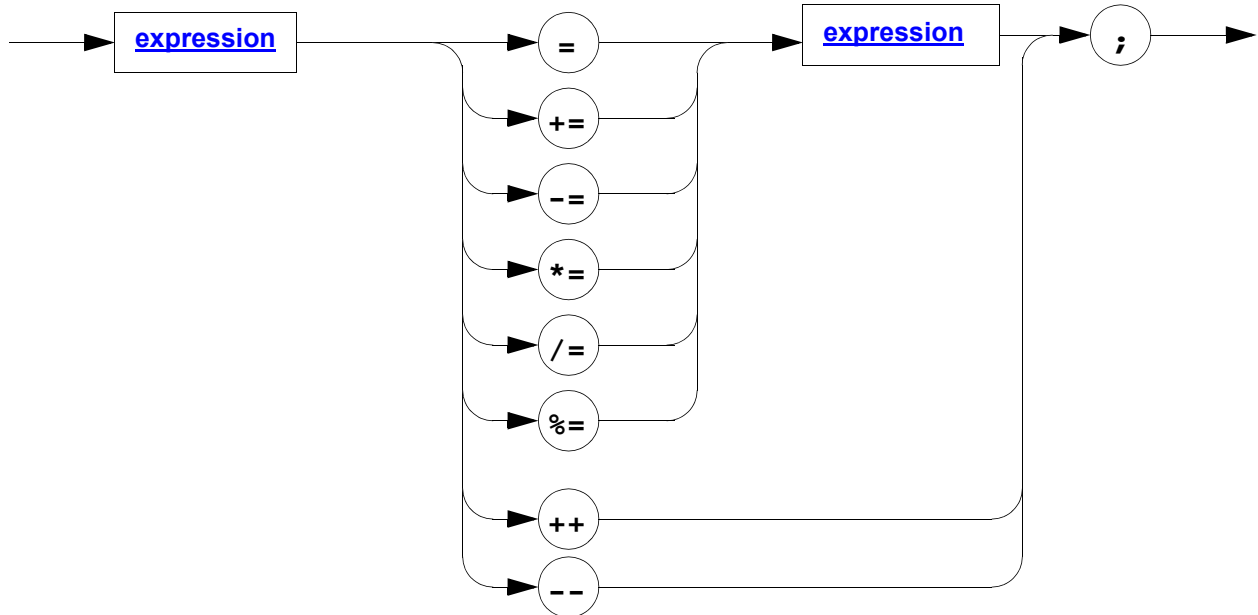
3.1.2 statement





3.1.3 assign_stmt

(assign statement)



This is the general form of an assignment. The first expression denotes a memory location, e. g. may be the name of a variable. The second expression may be as simple as a single literal, or as complex as required. Please see "Operator =" on page 43 for implicit type conversion.

Example:

```
x = 10;
str.f = x + 13;
```

The operator += increments the left expression by the value of the right expression.

Operator -= subtracts the value of the right expression from the left expression and stores it at the variable defined by the left expression.

The operators *= (multiplication and assignment), /= (division and assignment) and %= (modulo operation and assignment) work equivalent.

The operator ++ increments the value of the left expression (must be a variable) by 1. The operator -- decrements the value of the left expression.

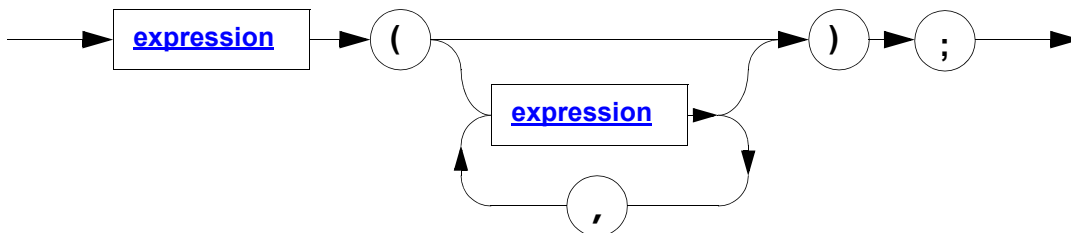
If += is used for string expressions the right string is concatenated to the left string variable.

Example:

```
i++;  
z--;  
string s="Hello ";  
s+="world ";  
k+=5;
```

3.1.4 call_stmt

(function call command)



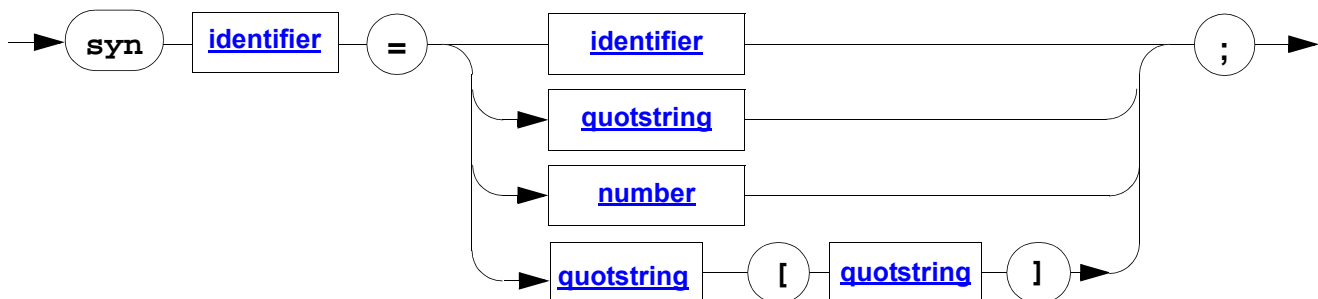
Defines the syntax of a function call, where the result is ignored or the result type is defined as void. For information about type compatibility of parameters see the description of `function_stmt`.

Example:

```
triangle(a, b, c);  
set_length(1);
```

3.1.5 syn_stmt

(syn statement)



Using the keyword `syn` you can define an alias name for an identifier, e. g. a class name.

Example:

```
syn CForm  = cblForm;
syn maxCnt = 4;
syn hello  = "Hello World!";
```

After this declaration the identifier CForm can be used equivalent to the identifier cblForm. Due to readability considerations of programs this statement should be used very restrictively!

The last syntax has been introduced especially for access of NLS-Texts for project H.O.T. where the first quotstring specifies the catalogue to be opened, the quotstring in brackets is the key of the text to be fetched from this catalogue.

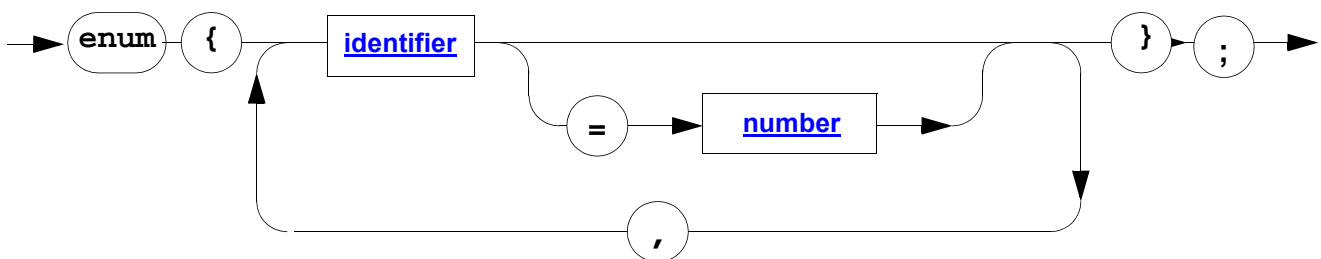
Example:

```
syn Catalog = "messages.txt";
syn msg1    = Catalog [ "MSG1" ];
```

In the example msg1 is a synonym for the quotstring which has been retrieved from the catalogue messages.txt with key "MSG1" by the H.O.T.-OLE-Component HOT.NLS.1.

3.1.6 enum_stmt

(enum statement)



Using the keyword `enum` you can define the enumeration of constants. The numeric values are automatically incremented starting at 0. Alternatively, a specific value can also be set.

Example:

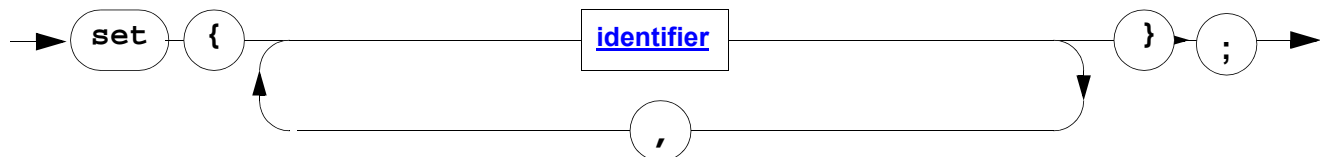
```
enum { mtWarning, mtError, mtInformation, mtConfirmation, mtCustom };

enum { FTPProto_RmxScsi=1,
      FTPProto_FTP=3,
```

```
FTPProto_FJAM=4,  
FTPProto_DispaList=6 };
```

3.1.7 set_stmt

(set statement)



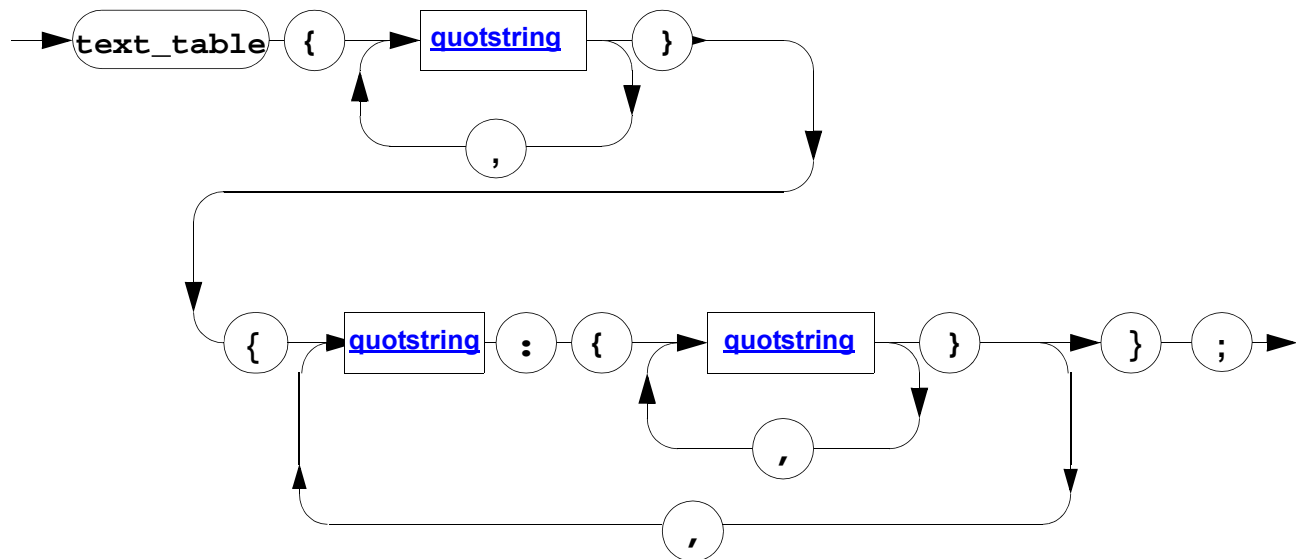
Using the keyword `set` you can define the enumeration of constants. The numeric values are automatically doubled starting at 1. The first identifier is assigned the value 1, the second the value 2, the third the value 4, the fourth the value 8, etc.

Example:

```
set { mbYes, mbNo, mbOK, mbCancel, mbAbort, mbRetry, mbIgnore,  
      mbAll, mbNoToAll, mbYesToAll, mbHelp };
```

3.1.8 text_table_stmt

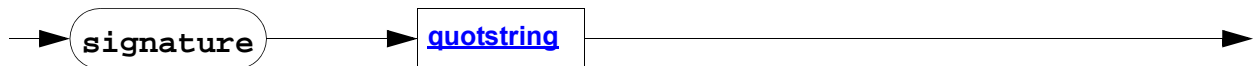
(text_table statement)



Text tables can be used to implement language independent CBL applications (see "*CBL - National Language Support*" on page 174 for further details).

3.1.9 signature_stmt

(signature statement)



Signatures are used to protect cbl files against unlicensed use and modifications. Signatures are generated by the CBL Interpreter itself and appended to a file, depending on the current user profile.

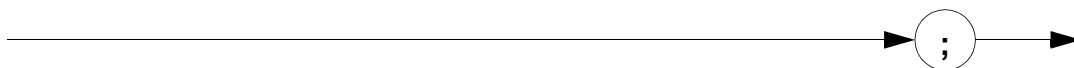
In the first release of HiPath 4000 Expert Access this mechanism will just be used to protect files against modification. It **could** be used to limit the capabilities to develop own CBL applications.

Do not modify system files secured with signatures! These files will be unusable and will not compile.

Signatures are appended automatically to the CBL source file. There may be more than one statements per file.

3.1.10 empty_stmt

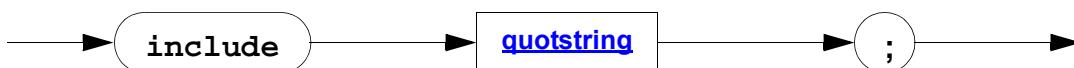
(empty statement)



The *empty_statement* is provided for convenience only. It allows additional semicolons between statements.

3.1.11 include_stmt

(include statement)



The *include_stmt* instructs the interpreter to read another source file in addition to the one that contains the *include_stmt*. The name of the file must be enclosed between double quotes. You can use both slash ("/") and backslash ("\") to separate directories. As always in quoted strings, if you use backslash you have to double every occurrence of backslash ("\\").

Example:

```
include "source_b.cbl";
```

You can use an absolute (e.g. C:/rootdir/subdir/inc.cbl) or a relative file path (e. g. source_b.cbl, subdir/source_c.cbl, ./subdir/source_c.cbl, ../subdir2/source_d.cbl) to specify the file. It is recommended to use relative file paths wherever possible. Thereby it's easy to move a CBL application to another directory or drive, as long as the relative position of the source files retains intact.

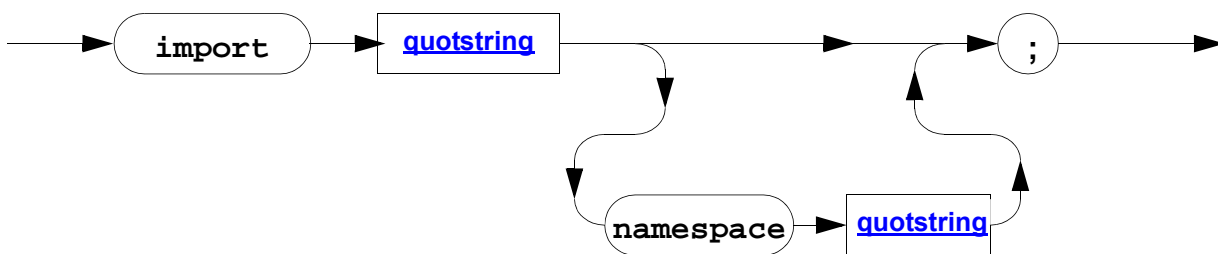
If you use an relative path, the file is searched in the following order:

1. The file is first searched in the system include directory. This path is determined during installation of HiPath 4000 Expert Access and can be checked in HiPath 4000 Expert Access-Options
2. Next the directory relative to the directory of the current source file is searched.

After determining the full path of the include file, the CBL Interpreter first checks, if this file already has been included previously. If this is the case, the second include statement is ignored and thereby problems due to duplicate definition of classes etc. are avoided. So this feature supports modular development of subsystems.

3.1.12 import_stmt

(import statement)



Using the *import* statement the complete OLE interface of an OLE automation server can be imported. The *import* keyword either follows the class string under which the OLE server is registered or the absolute file name of its type library (.tlb file).

If the passed name ends with "wsdl" the name is interpreted as URL or file name of a WSDL description of a WebService. See section 3.3.5, "Working with the SOAP Interface", on page 62 for details.

If the passed name has the suffix “xsd” it is interpreted as file name of a XML Schema Definition File. Please refer to section 3.3.6, “Import of XSD Files”, on page 64 for details.

If the passed name ends with “asn”, it is interpreted as file name of a ASN.1 module. If the suffix is “zip”, the CBL interpreter unpacks this file and expects a file with suffix “lst” (with the same filename as the .zip-File) that should give the order for reading the ASN.1 modules. See section 3.3.7, “Import of ASN.1-Modules”, on page 67 for details.

By specifying the optional "namespace", the relevant prefix can be assigned to each OLE class published by the OLE server.

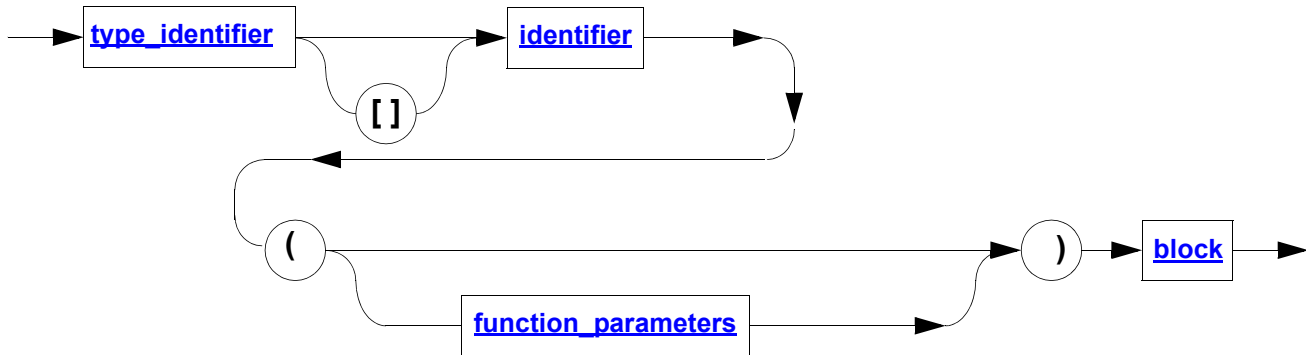
The classes, functions and property attributes that were made available with the import statement can be displayed in the CBL Interpreter's data inspector. A description of meaning and action is provided in the documentation provided with the relevant application.

The following example shows how the OLE interface can be imported and used by MS Excel:

```
import "Excel.Sheet" namespace Excel_  
  
Excel_Application excel;  
  
Excel_Workbooks workbooks;  
  
workbooks=excel.Workbooks;  
  
workbooks.Add();  
  
excel.Range("A1").Formula="Hello World";  
  
Excel_Workbook workbook=excel.ActiveWorkbook;  
  
workbook.SaveAs("c:\\temp\\hello.xls");  
  
workbook.Close();
```

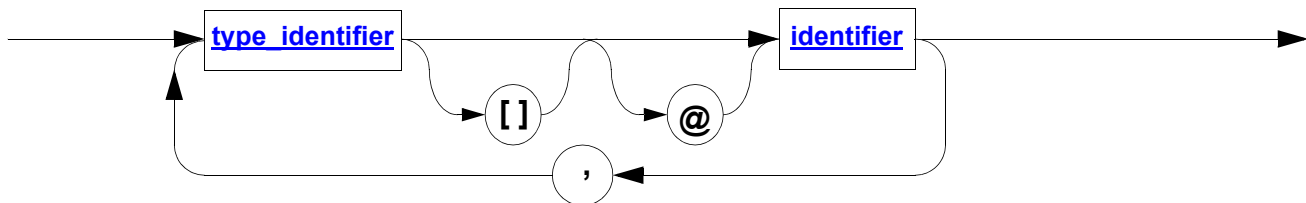
3.1.13 function_stmt

(function declaration)



3.1.14 function_parameters

(function parameters)



The *type-identifier* specifies the type of data the function returns. It can be defined by a class or be of the type `int`, `string`, `variant` or `void` if no value is returned. It is also allowed to return an array. The type `variant` is used only in the context of OLE automation.

The second *identifier* is the name of the function. It is followed by a parameter list, which is a comma separated list of variable names (*identifier*) and their associated types (*type identifier*) that receive the values of the arguments when the function is called.

By default parameters are passed by value, i. e. the function gets a local copy of the parameters. Modifications of the parameters within the function are not seen by the caller. In contrast, the `@` sign denotes a reference parameter. Reference parameters don't receive copies of the arguments, but use the same memory location as the arguments. Modifications of the parameters within the function are therefore seen by the caller. Also, the current argument for a reference parameter must always denote a memory location, e. g. be a variable, and must not be an arbitrary expression.

For compatibility of actual and formal parameters the same rules are valid as for assignment (see there). Therefore it is possible to pass an `int` variable to a `string` parameter and vice versa. This is even true for reference parameters!

The parameter list of a function is optional, but the parentheses are required. The *block* is the body of the function, which consists of the code. A function's code cannot be accessed by any statement in any other function except through a call to that function. Unless the function uses global variables or data, the code of its body can neither affect nor be affected by other parts of the program.



Although it is possible to define functions within functions this feature is not available in the interpreter.

Example:

```
int sqr(int x)
{
    x = x * x;
    return (x);
}

int a;
int b;
a = 3;

b = sqr(a);
// b is 9; a still is 3

b = sqr(4);
// b is 16
```

Example:

```
string concat(string a, string b)
{
    return a + b;
}
```

Example:

```
void func_call()
{
    int a;
    a = 10;
    z = sqr(a);
}
```

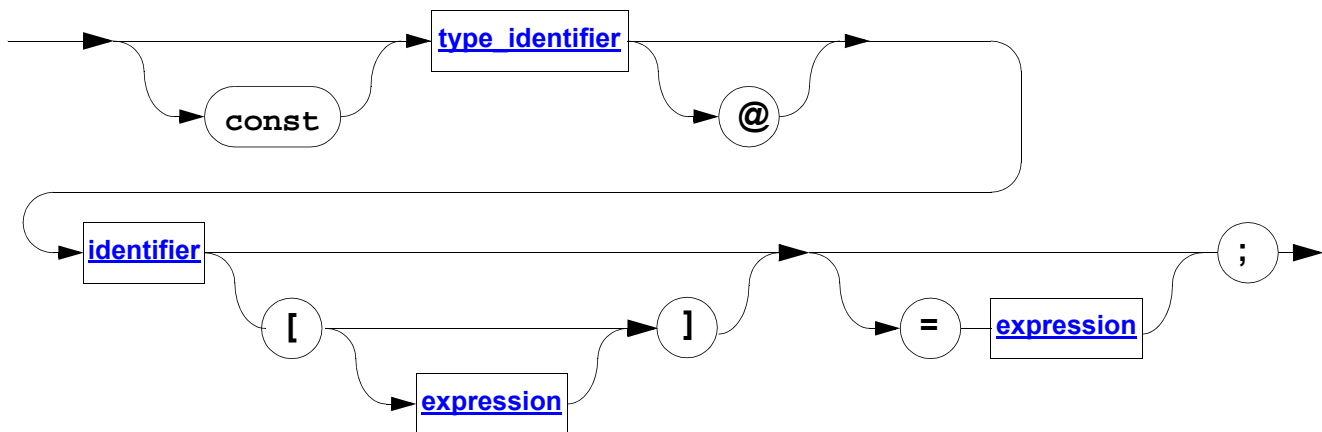
Example: Call by reference

```
void append(string[] @list, string txt)
{
    int cnt=list.count;
    list[cnt]=txt;
}
```

```
string [] list;  
append(list, "aaa");  
append(list, "bbb");  
append(list, "ccc");
```

3.1.15 **declaration_stmt**

(Variable statement)



Variables are declared in four different places:

- inside blocks (inside or outside of a function)
- in the definition of function parameters
- inside a class, but outside of functions
- outside of all classes and functions.

These are termed local variables, formal parameters, member variables and global variables respectively.

With the *declaration_stmt* you are declaring local variables, member variables and global variables (It depends if the declaration statement is inside a block, or outside of all blocks). Local variables may be referenced only by statements that are inside the block in which the variables are declared or in a block, nested within that block.

The *type-identifier* defines the type of the variable. The *identifier* is the name of the variable. Variables, class members and array elements are automatically initialized according to their type: Integer variables with 0, string variables with an empty string (""), reference variables with the predefined constant `null`.

A '@' in front of the *type-identifier* can be used to define a reference variable. This is a variable that does not contain a value itself but it "points" to another variable/object. See Section 3.3.1 on page 3-51 for further details.

To declare an *array* the variable name has to be followed by brackets.

There are two different methods to define arrays (with/without upper bound) as well as two different types to access array elements (string indexed, integer indexed). Memory allocation for an array is always done when a new array element is accessed, not when the array is defined. If one tries to access an element by an index and there is not yet an element, an "empty" element is created automatically.

`string list[10];` defines an array with a maximum of 10 elements. This **boundary** is checked with every access to this array. Note, that the expression in brackets need not be an constant expression! It is evaluated once, when the array declaration statement is executed. The array may be either string indexed or integer indexed.

`string list[];` without an expression in brackets defines a **dynamic array** of strings. Dynamic means, that there is practically no maximum number of elements. The array may be either string indexed or integer indexed.

The first access to an array defines the type of indexing. If the first access was done with an **integer index**, this array can *only* be accessed with integer indices. It is allowed to have gaps within the index values, the elements in-between are allocated automatically. Integer indices always start with 0. The following example creates two strings in the array called list.

```
list[3]="Gerry";
```

```
list[6]="Bernd";
```

The index range is from 0 to 99999.

If the first access to an array (e.g. `string[]`) was done with a string index, this array gets **string indexed**. The elements of a string indexed array are automatically sorted alphabetically due to performance reasons. Example:

```
string phone[];
phone["Erich"] = "18492";
phone["Roland"] = "34928";
phone["Alexandra"] = "23049";
```

String indexed arrays can be accessed both with string or integer indices, however you cannot allocate an additional array element using an integer index for an array that uses string type indices.

Each array variable has a property *count*, which holds the number of array elements currently allocated, i.e. the array elements can be accessed using indexes 0..*count*-1. Example:

```
int i;
for i=0 to phone.count-1 // will loop 3 times in the example above
{
    system.ConsolePrint(phone[i]);
}
```

This example prints the values "23049", "18492" and "34928" in the console window (sorted alphabetically by *string index*).

```
int i;
for i=0 to list.count-1 // will loop 7 times in the example above
{
    system.ConsolePrint(list[i]);
}
```

This example prints the values "", "", "", "Gerry", "", "", "Bernd" in the console window.

Currently only one-dimensional arrays are supported. If you absolutely need multi-dimensional arrays, you can embed an array into a class and build an array of this class.

The optional `= expression` clause is used to define an initial value for a variable or symbolic constant. It is a shorthand notation for definition of a variable and a subsequent assignment.

The initialization of member variables is effective whenever a new object instance of the class is created: Every new object's data variables are initialized with the values specified in the class declaration (see `class_member` below).

Note that class members of type reference cannot be initialized in the class declaration with a value other than `null`.

The `const` modifier is used to define symbolic constants, i. e. you define a variable who's cannot be modified later on. For `const` variables you *must* specify a initialization expression, because there is no other way to assign a value to a `const` variable.

Example:

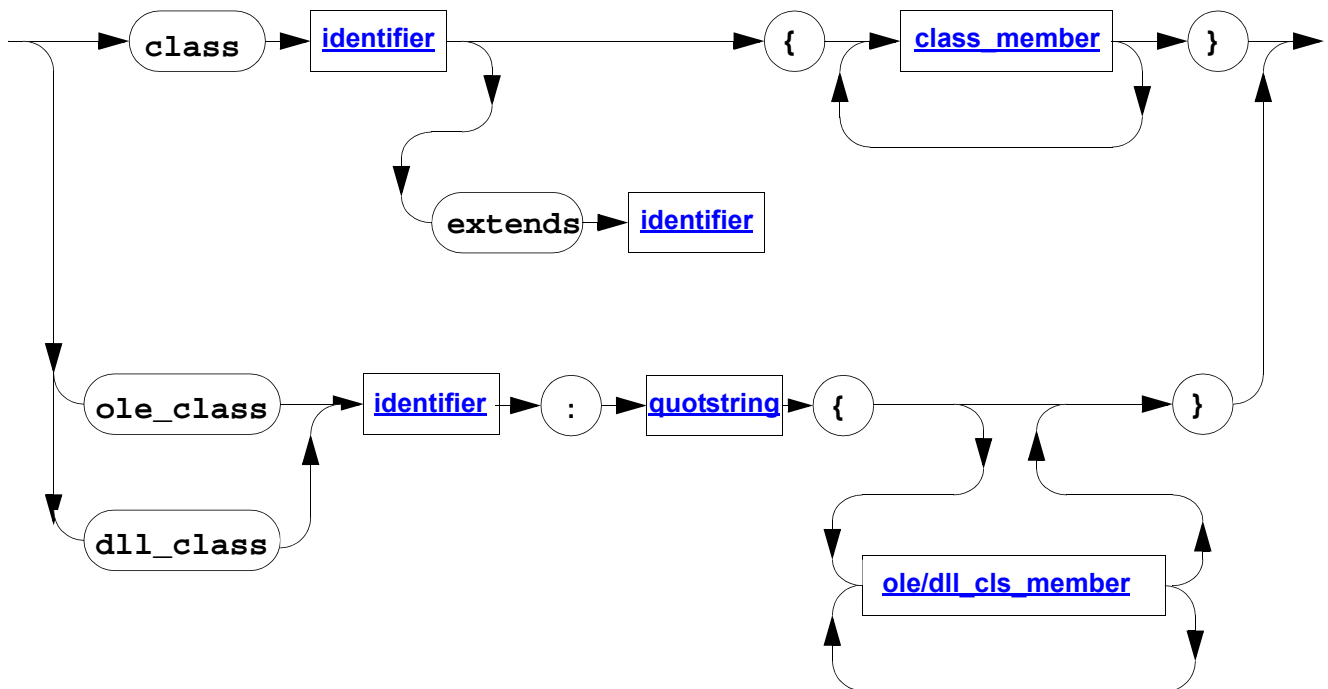
```
int a;
...// assign some value to variable a
string chars[a]; // fixed sized array of strings

int b[];
int c[10]= {"Bernd":39153, "Erich":31562, "Roland":34752}; // fixed
           // sized string indexed array of ints
int d[10]= {89, 1199, 3}; // fixed sized integer indexed array of ints

string name;
const int True=1;
const string name="Hugo";
```

3.1.16 class_stmt

(class statement)



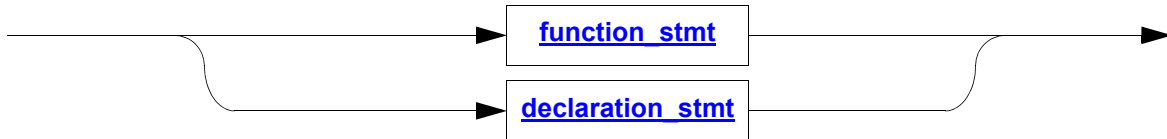
A class declaration defines a new type that links code and data. Classes are created using the keyword `class`. The *identifier* specifies the name of the class. The code and data of classes are enclosed in braces.

A class can be derived from a base class using the keyword `extends`. The new class inherits all members (attributes and methods) of the base class. If a method/attribute has the same name as a base class member, only the member of the derived class will be accessible (overriding).

OLE classes are described in chapter 3.3.3, "Using the OLE interface".

DLL classes are described in chapter 3.3.4, "Using the DLL interface".

3.1.17 **class_member**



A class member can either be a *function_stmt* or a *declaration_stmt*.

Example:

```
class book {
    string author;
    string title;
    string keywords[];
    int number;
}
```

In the case above, the class consists only of declarations. The data members of a class or object are called *attributes* in CBL.

This example below is more complex. The class consists of declarations and a function. *employee* is a simple class, which is used to store a name and an address. The function *tax* is used to compute an employee's wage. In this special case *wage* will be 3000.

```
class employee {
    string name;
    string address;
    int tax(int pay)
    {
        return pay - 2000;
    }
}
```

```
...
void foo()
{   employee fred;
    int wage;
    fred.name = "Frederick";
    fred.address = "Munich";
    wage = fred.tax(5000);
    ...
}
```

In the example below, an initialization for data members is given:

```
class Point
{   int top  = -1;
    int left = -1;
}

Point p1; // p1.top now is -1 and p1.left is -1
Point p2 = {left: 10}; // p2.top now is -1 and p2.left=10;
```

Example for class hierarchy:

```
class base
{   string a;
    int    b;
    int    c;
    int    f()    {   return b+c;   }
    void    g()    {   system.ConsolePrint("base::g called");   }
}
class subclass extends base
{   string c;
    string d;
    void    g()    {   system.ConsolePrint("subclass::g called");   }
    void    h()    {   system.ConsolePrint("subclass::h called");   }
}
base A;
subclass B;
A.f();
A.g();
B.f();
B.g();
B.h();
```

In this example a base class "base" and a derived class "subclass" is defined. The subsequent calls of the methods produce the following output in the console window:

base::g called
 subclass::g called
 subclass::h called

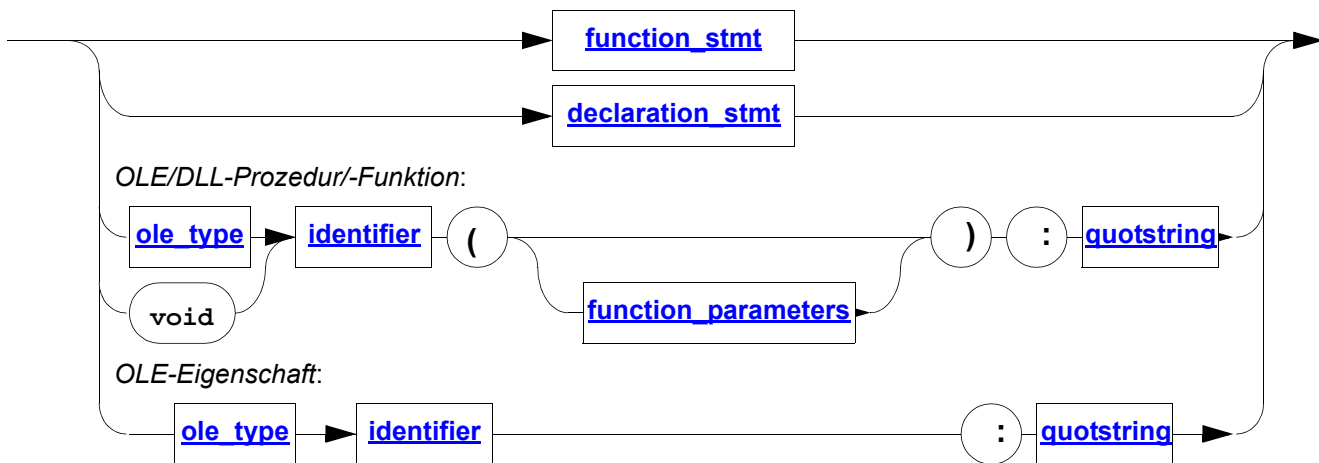
The function f() is inherited by the subclass. Please note that f always accesses the attributes b and c of class "base".

Function g() is overridden by the derived class, so the function g() of the subclass will be executed for objects of type subclass.

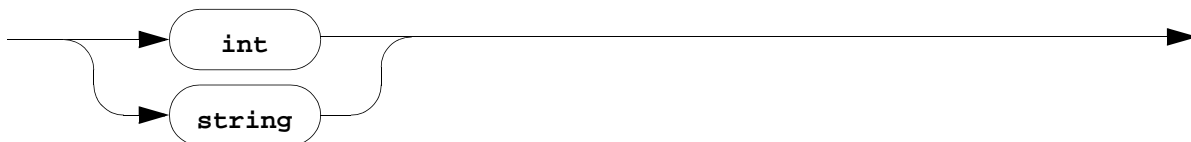
Function h() is an additional method defined by the subclass.

The member c of the base class cannot be accessed directly by a method of subclass because it is hidden by "string c". It can be accessed using the predefined "pointer to base class" called super.

3.1.18 ole/dll_cls_member



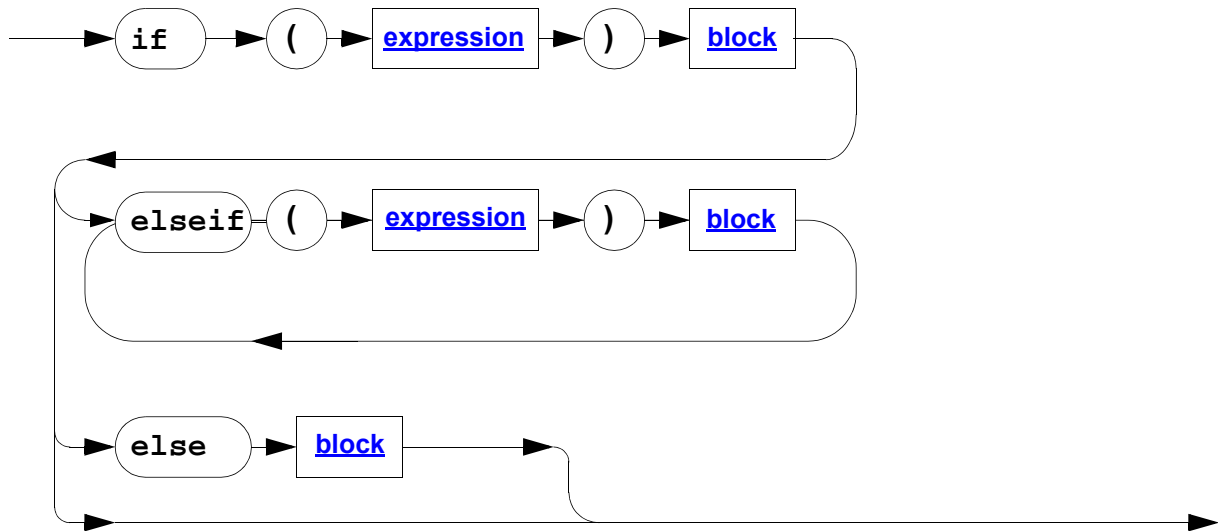
3.1.19 ole_type



OLE classes can additionally contain "externally" implemented functions and properties. The types for parameters, return value and properties is limited to int and string. The quotstring behind the colon specifies the name of the OLE procedure, function or property as published by the OLE server application.

3.1.20 if_stmt

(if statement)



If the *if-expression* is true (anything other than 0), the corresponding block will be executed. Otherwise the *elseif-expressions* are tested one after the other until one is found which yields true. Then the corresponding block is executed. If none yields true, the block which is the target of *else* will be executed, if it exists.

Please note that the condition expressions can be as complex as you want using the boolean operators "and" (&&), or (| |) and not (!).

Example:

```

x = 10;
if (x < y)
{
    x = y;
}
elseif (x > y)
{
    y = x;
}
else
{
    x = x + 1;
    y = y + 1;
}
  
```

Example:

```
x = 22;  
y = i;  
if (x == y)  
{  
    y = y + 1;  
}
```

Expression must yield an integer result. Here `if (x == y)` means: If `x` is equal `y` the following block (in the braces) will be executed. `x = y` would be an assignment, which does not return a value and therefore would cause a runtime error.

Hint: Could/Should be detected at compile time!

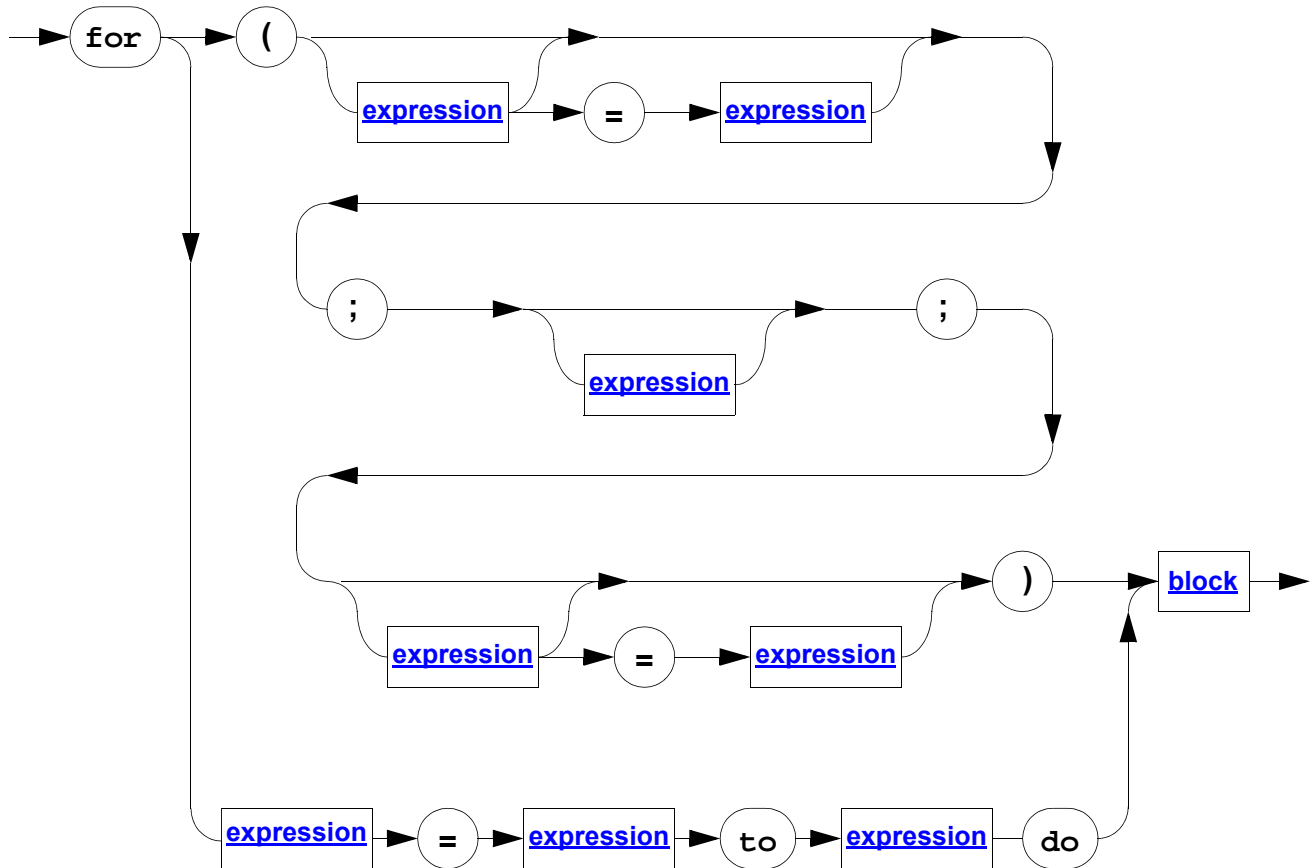
Example:

```
if (x)  
{  
    ...  
}
```

If `x` is anything other than 0, the block will be executed.

3.1.21 for_stmt

(for statement)



The keyword *for* is followed by three expressions - each of them is optional - enclosed in parentheses. In general the first expression is an assignment statement that is used to set the loop control variable, but may also be e. g. a function call. The second is a relational expression that determines when the loop exists. The third is an increment that defines how the loop control variable changes each time the loop is repeated. These three sections are separated by semicolons. The *for-loop* continues to execute the block as long as the condition is true. Due to performance reasons there is also a second possibility to express a *for_stmt*. This form of the *for_stmt* consists of an assignment and a limitation. Therefore the first expression has to be a variable, the second and third expression have to be numerical. The increment is always 1 (see example above).

Example: for loop 1

```
for (x = 7; x < 2; x = x - 1)
{
    sum = sum * x;
}
```

Example: for loop 2

```
for i = 1 to 10 do
{....}
```

Example: for loop 3

```
for (;;)
{...}
```

This loop will run forever.(Actually this is not guaranteed, because a `break` statement inside in the block, causes immediate termination.)

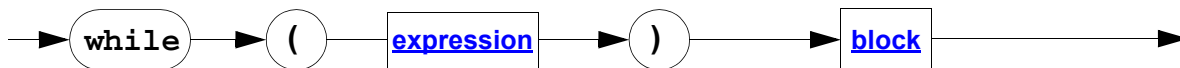
Example: for loop 4

```
x = 6;
for ( ; x < 10; )
{ x = x + 1; }
```

The initialization section has been left blank, and `x` is initialized before the loop is entered.

3.1.22 while_stmt

(while statement)



The expression enclosed by the parentheses after the keyword `while` must be a condition. True is represented by any nonzero value. The *block* is iterated while the condition is true. When the condition becomes false (`==0`), the program passes to the statement after the block.

Example:

```
string answer;
string text;
answer = "N";
while (answer != "Y")
{
```

```
    answer=question(text);  
}  
...
```

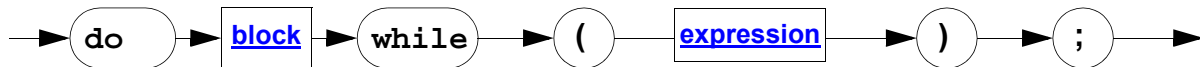
The function `question(text)` (which must be defined anywhere in the program code) will be iterated as long as "Y" will be returned.

The next example has the same effect:

```
string answer;  
while (question(answer) != "Y") { }  
...
```

3.1.23 do_while_stmt

(do_while statement)



Unlike for and while loops, which test the loop condition at the top of the loop, the *do_while_stmt* checks its condition (the expression enclosed by the parentheses) at the bottom of the loop. That means that the block is always executed at least once. The *do_while_stmt* iterates until the condition becomes false.

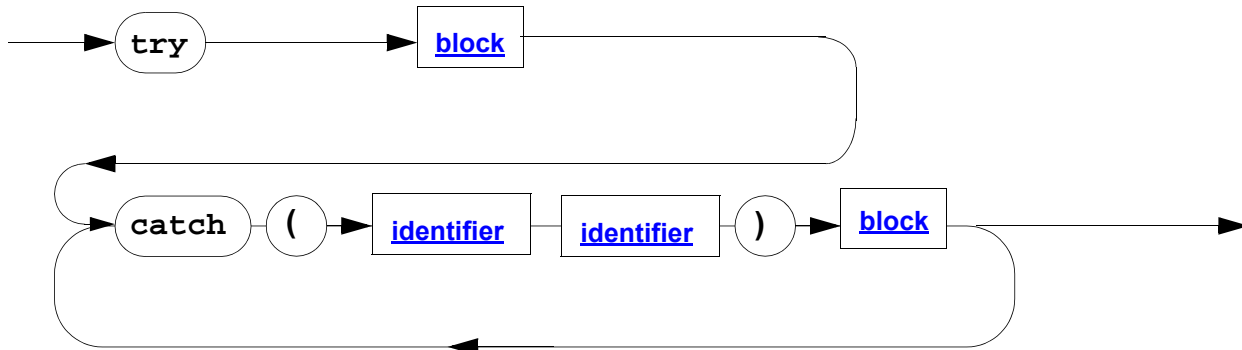
Example:

```
string answer;  
string text;  
answer = "Y";  
do {  
    answer=question(text);  
} while (answer != "Y")  
...
```

Although `answer` is "Y" the *block* after `do` is executed. It depends on the return value of the function `question(text)` if it is repeated or not, no matter which value `answer` has before the *do_while_stmt*. (In contrast to the *while_stmt*.)

3.1.24 try_catch_stmt

(try_catch statement)



Using the try_catch statement in conjunction with the throw statement you can implement exception handling as in C++ or Java.

If an exception object is "thrown" to a random call hierarchy within the try block, then it can be "caught" via subsequent catch instructions. An exception object is caught if its type, i.e. the exception class, matches; in other words, the thrown object contains the class to be caught in its class hierarchy (see *"class_stmt" on page 25*, see keyword `extends`).

The identifiers that follow the keyword `catch` specify an exception class and the name of the exception object under the following block.

For an example of exception handling see *"throw_stmt" on page 34*.

3.1.25 throw_stmt

(throw statement)



Exception handling can be triggered with the throw instruction.

The keyword `throw` follows the name of an object that is derived from the predefined "Exception" base class (see *"class_stmt" on page 25*, see keyword `extends`).

All statements following the `throw` statement are ignored until the object is "caught" by a suitable `catch` statement (see *"try_catch_stmt" on page 34*).

If a suitable catch statement was not found, the CBL program is terminated with an "Unhandled Exception" runtime error.

The base class of all exceptions is predefined in the system and is called "Exception". Moreover, an OleException exception class that is triggered as soon as a runtime error occurs in response to OLE problems is predefined:

```
class Exception
{
    string File;
    int    Line;

    string Message;
}

class OleException extends Exception
{
}
```

The properties `File` and `Line` are automatically set by the CBL Interpreter on the execution position of the throw statement. The origin of the exception can thus be determined later.

Example: Exception handling

```
try
{
    for i=1 to 100 do
    {
        comwinFrame.ConsolePrint(_string(i));
        if (i==10)
        {
            Exception e1;
            e1.Message="Abrupt cancellation";
            throw e1;
        }
    }
}
catch(Exception e)
{
    comwinFrame.ConsolePrint(e.Message);
}
```

The loop is quit after the 10th run. Information on the exception can be passed on via the exception object.

3.1.26 with_stmt

(with statement)



The *with_stmt* has to be followed by an expression enclosed by the parentheses and a block. The expression must denote a memory location. This statement makes it easier to access data and functions of a class.

Example:

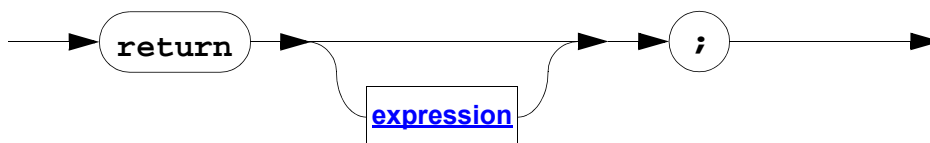
```
class xyz
{
    int a
    int b
    void m()
    {...}
    ...
}
xyz ABC
...
with (ABC)
{
    funct(a);
    b = 5
    m();
    ...
}
```

Without the *with_stmt* the code after the "..." had to be:

```
funct(ABC.a)
ABC.b = 5;
ABC.m()
```

3.1.27 return_stmt

(return statement)



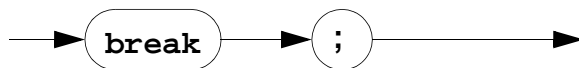
The *return_stmt* was already used in some examples above.

This statement is used to return from a function. It "jumps" back to the point at which the call of the function was made. If *return* has a value associated with it, the value of *expression* becomes the return value of the function. A non-void function must return a value.

You can use as many *return* statements as you like within a function. However, the function will stop executing as soon as it encounters the first *return*. The *}* that ends a function also causes the function to return. It is the same as a *return* without any specified value.

3.1.28 **break_stmt**

(break statement)

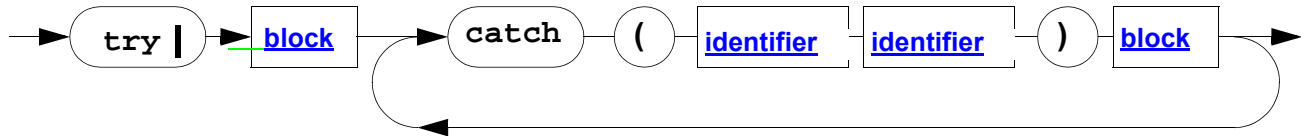


The *break_stmt* is used to terminate a loop (*for*-loop, *while*-loop, *do-while*-loop) immediately, bypassing the normal loop conditional test. When the *break* statement is encountered inside a loop the program resumes at the next statement following the loop. A *break* causes an exit only from the innermost loop.

Example:

```
for (t = 0; t < 100; t = t + 1)
{
    count = 1;
    for (;;)
    {
        count = count + 1;
        if (count == 10)
        {
            break;
        }
    }
}
```

| 3.1.29 .try_catch_stmt



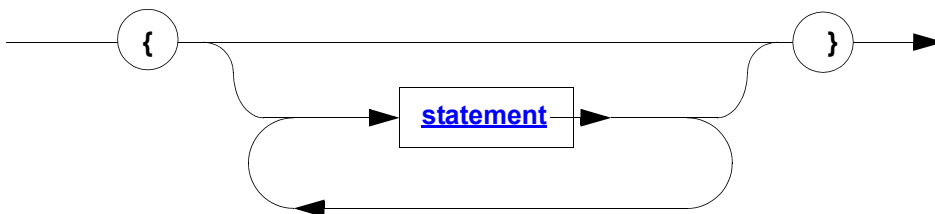
The try-catch-statement can be used to catch exception objects that have been thrown by means of the throw statement. The first identifier specifies the exception class, the second the exception object). If there is no exception the exception block is ignored.

Example:

```
try
{
    g();
}
catch (Exception e)
{
    comwinFrame.ConsolePrint("exception caught");
}
```

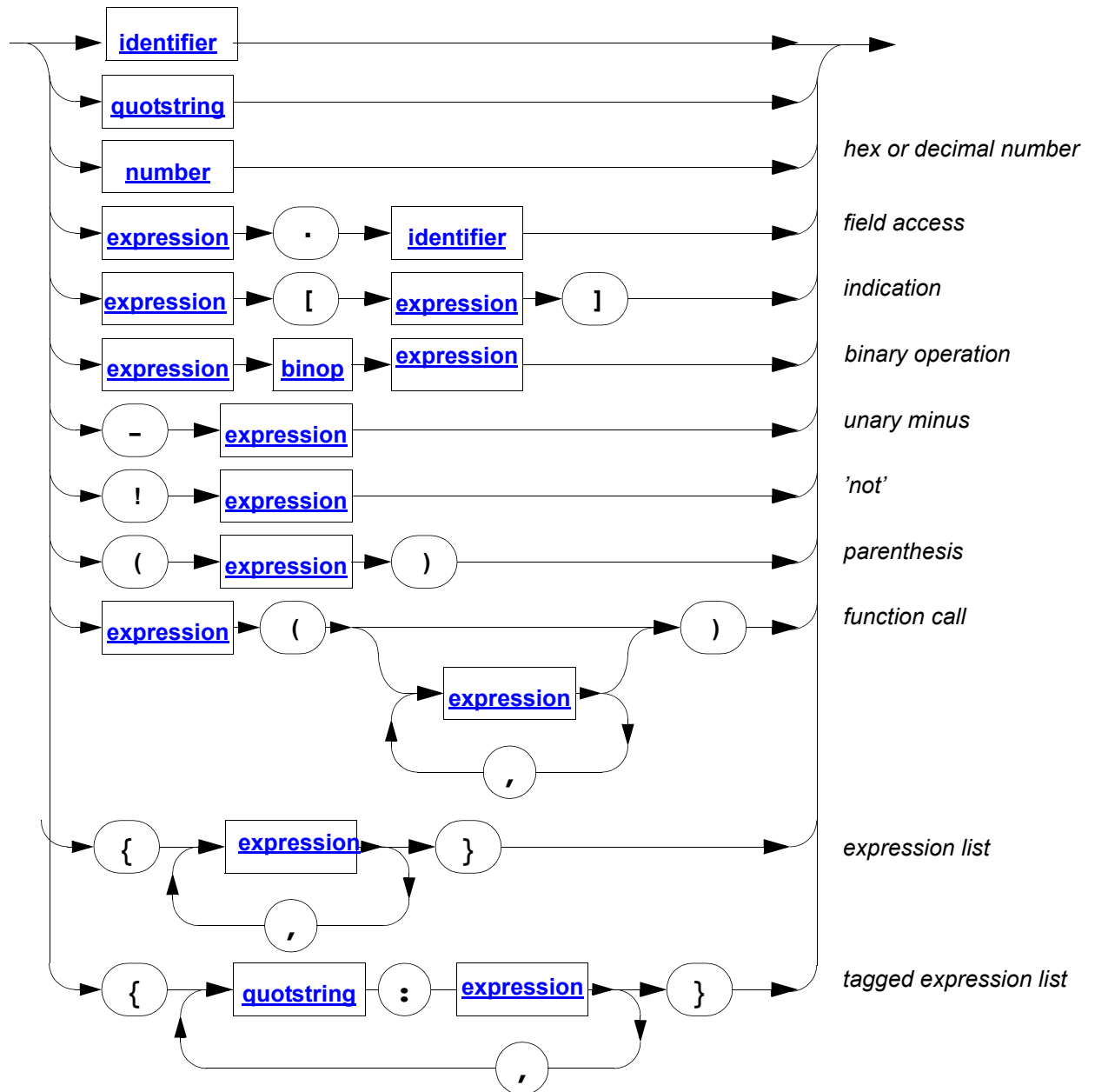
| 3.1.30 block

(block)

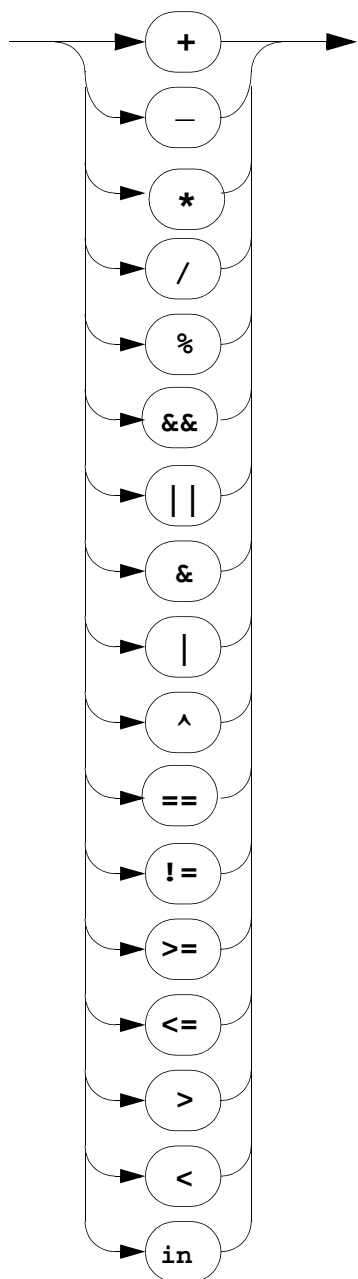


A block consists of none, one or more statements. A block begins with an opening curly brace and ends with a closing curly brace.

3.1.31 expression



3.1.32 binop

	addition / string concatenation	see Operator +
	subtraction	see Operator -
	multiplication	
	division	
	modulus	
	logical AND	
	logical OR	
	bitwise AND	
	bitwise OR	
	bitwise exclusive OR (XOR)	
	equal	
	not equal	
	greater than or equal	
	less than or equal	
	greater than	
	less than	
	containment operator	see Operator in

There are some restrictions concerning the operand types allowed for the different operators: Arithmetic, logical and bitwise operators are defined for int operands only. The plus sign is also used for string concatenation (see table below).

The comparison operators ==, <, >, <=, >=, != are defined for the basic types int and string.

The following tables give you an overview about the possibilities of combinations with the four operators "+", "-", "in" and "=".

(If there is a „-“ in a cell, it means that this combination is not defined.)

Operator +

+		right operand			
left operand		int	string	class	[]
	int	arithmetic addition	-	-	-
	string	-	string concatenation (1)	-	-
	class	-	-	-	-
	[]	-	-	-	-

(1) concatenates one string onto the end of another

Example:

```
string s1;
string s2;
string hello;
s1 = "Hi";
s2 = " there!";
hello = s1 + "," + s2;
```

The string hello will be: "Hi, there!".

Operator -

-		right operand			
left operand		int	string	class	[]
	int	arithmetic subtraction	-	-	-
	string	-	-	-	-
	class	-	-	-	-
	[]	-	-	-	-

Operator in

in		right operand			
left operand		int	string	class	[]
	int	-	-	-	-
	string	-	checking occurrence (1)	-	searching element in string[] (2)
	class	-	-	-	
	[]	-	-	-	-

(1) Example:

```
string hello;
...
if ("he" in hello)
{...}
```

If "he" occurs in the string hello, the expression ("he" in hello) is true and the block {...} will be executed.

(2) Example:

```
string hello;
string Text[];
...
if (hello in Text)
{...}
```

If the string in variable `hello` is an element of the array `Text`, the expression `(hello in Text)` is true and the block `{...}` will be executed.

Operator =

=		right operand			
left operand		int	string	class	[]
	int	assignment	string - int conversion (2)	-	-
	string	int - string conversion (1)	assignment	serialization for storage (3)	
	class	-	serialization for storage (3)	assignment	-
	[]	-		-	assignment

(1) converts an integer into a string.

Example:

```
string hello;
int x;
x = 10;
hello = x;
```

The 'result' of `hello` will be the string "10".

(2) converts a string into an integer. The conversion stops at the first non-interpretable character.

Example:

```
string hello;
int x;
hello = "10A";
x = hello;
```

The 'result' of `x` will be the integer 10.

(3) see later

Some examples referring to *expression*:

```
x = -3
x == -3 || x >= 0
x * -23 + 44 / 2
x = 5 / 2
```

x will be 2, because all numeric variables are integer (the remainder will be truncated)

```
x = 5 % 2
```

x will be 1, the remainder of the integer division. The semantics of modulo is the same as in C.

```
hello = Text
```

The precedence of all operators is:

Highest	.
	[
	(
	* / % in
	+ -
	< <= > >=
	== !=
	&
	^
	&&
	!
Lowest	

Examples for expression lists:

```
class complex
{
  int re;
  int im;
}
// tagged expression list:
z={ im:18, re:1 };
x={ im:99 }; // x.re is set to it's default value (0)

// expression list:
int list[] = { 10 , 25, 30 };
```

An integer indexed array is initialized/assigned with an non-tagged expression list, e.g.

```
list={"Erich", "Bernd", "Roland"};
```

A String indexed array can be initialized/assigned a value with the tagged-expression-list .Example:

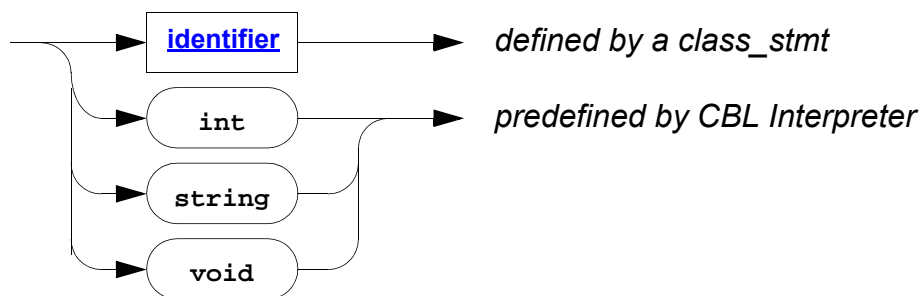
```
string phone[]={"Erich": "18492", "Roland": 34928, "Alexandra": "23049"};
```

In general, a tag can be either the identifier of a data member of a class or a string, which is used as index in an string indexed array.

Basically using an expression list in an assignment is simply a short-hand notation for multiple assignments.

An assignment to a class sets the fields which are named by the tags where the remaining fields get their initial value.

3.1.33 type_identifier



The *type-identifier* defines the type of a variable or function. You cannot define a variable of type void. The type void is used only as return type of functions, which do not return a result and as possible base type of references.

There is an additional predefined type *variant*, which is used only in the context of OLE automation.

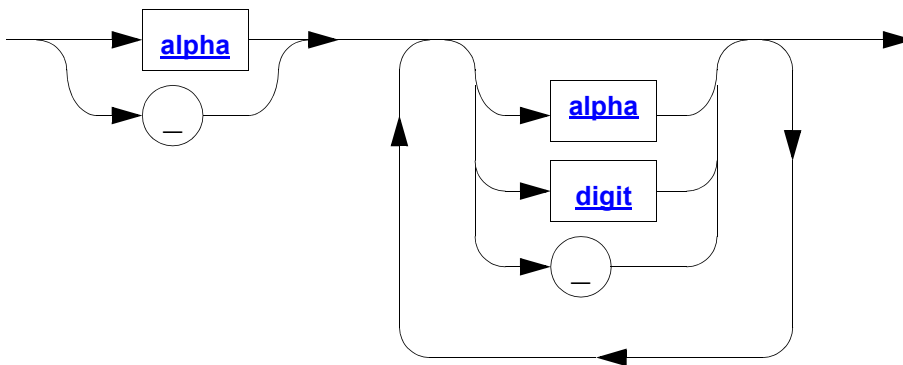
Example:

```
int i;  
void func();  
string Text[];  
string book(string title) {...}
```

An identifier which is defined by a *class_stmt*:

```
....  
class myclass {  
  ...  
}  
myclass data;  
....
```

3.1.34 identifier



The names of variables, functions, and other user-defined items are *identifiers*. These *identifiers* can vary from one to several characters. The first characters must be a letter or an underscore, and subsequent characters must be either letters, digits, or underscores. In an identifier, upper- and lowercase are treated as distinct. Hence, `x` and `X` would be two separate identifiers.

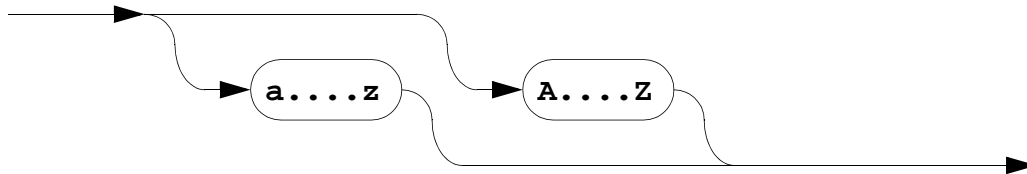


All names starting with an underscore are reserved for the CBL system and should therefore not be used in applications.

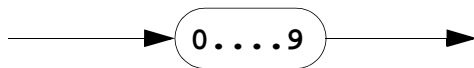
Example:

```
Correct:      _  
              test1  
              list_book  
Incorrect:    lx  
              hello!you  
              list...book
```

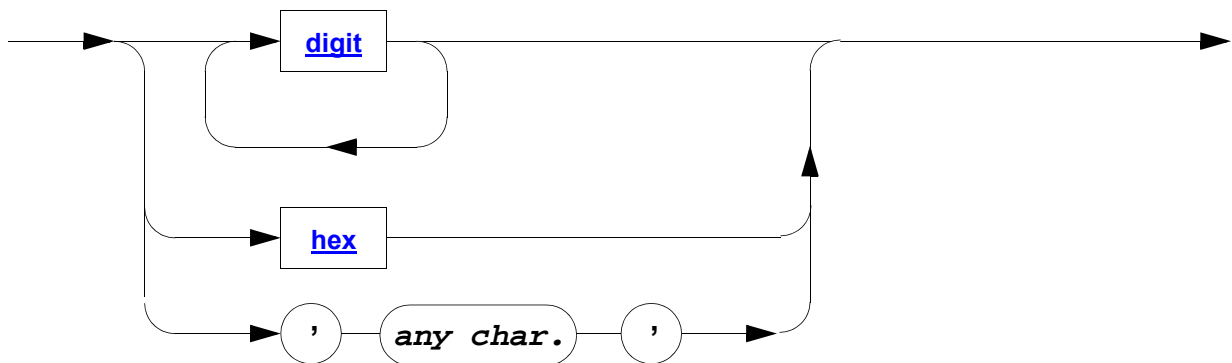

3.1.35 alpha



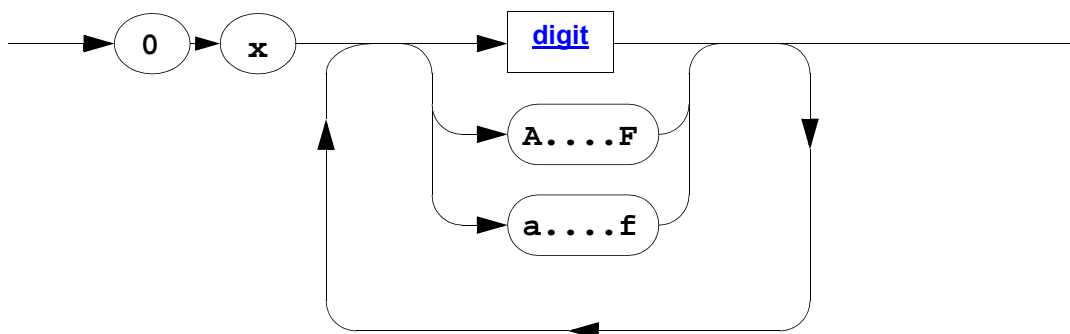
3.1.36 digit



3.1.37 number



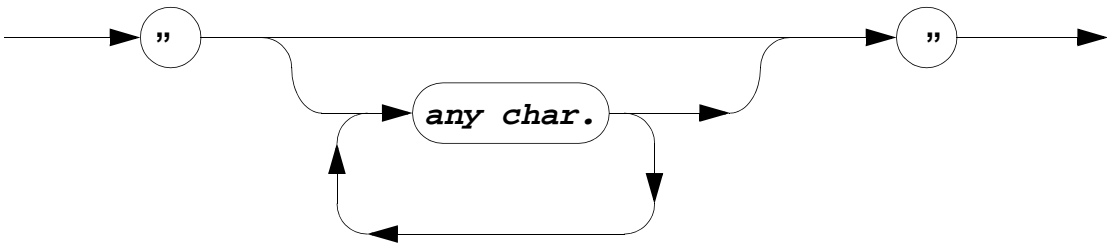
3.1.38 hex



Example:

```
int a;  
int b;  
a = 0x3F;      // 77 in decimal  
b = a + 0x64;  // equal to: b = a + 100
```

3.1.39 **quotstring**



A *quotstring* is a set of (printable) characters enclosed in double quotes.

Example:

```
"This is a 'simple' test"
```

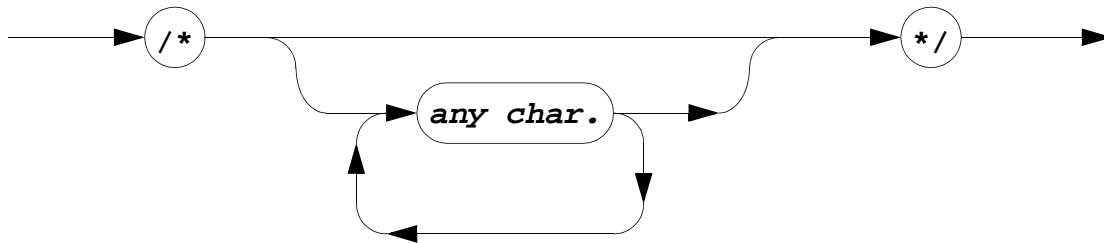
An exception is the backslash character (\). This character is used to include special codes in a string.

The backslash codes are:

\n	new line
\t	horizontal tab
\r	carriage return
\"	double quote
\\	backslash itself
\x	The following two characters are interpreted as hex digits. The corresponding ASCII character is inserted into the string (e. g. \x0a = newline character).

If a single backslash followed by any other character occurs in a string, the backslash will be ignored.

3.1.40 comment (structured)



Structured *comments* begin with the character pair `/*` and end with `*/`. There must be no spaces between the asterisk and the slash. The interpreter ignores any text between the beginning and ending comment symbols. Comments can be anywhere in a program, as long as they do not appear in the middle of a keyword or identifier or quoted string.

Example:

```

x = 10;
/* this is a comment */
x = x + /* add a number */ 5;

```

Comments may not be nested. That is, one comment cannot contain another comment.

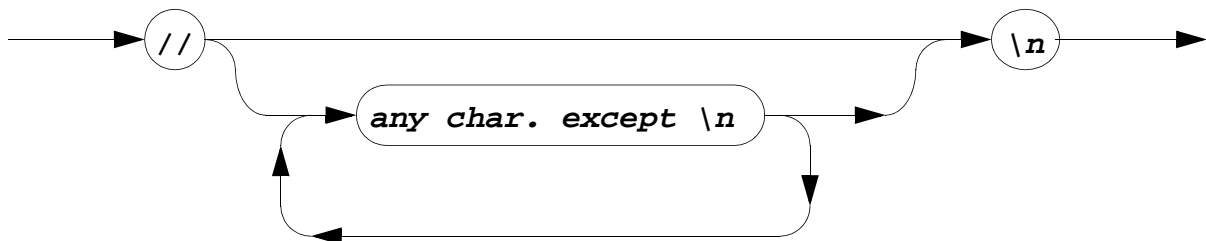
Example:

```

/* This is /* not allowed */ and causes an error */

```

3.1.41 comment (single line)



A *single line comment* begins with `//` and it ends at the end of the line. Whatever follows the `//` is ignored by the interpreter until the end of the line is reached.

Example:

```
x = 10;  
//this is a single line comment  
x = x + 5; //add a number
```

3.2 Summary of keywords

break	used to quit a for/do/while loop
catch	see <i>"try_catch_stmt" on page 34</i>
class	see <i>"class_stmt" on page 25</i>
const	used to define constants; see <i>"declaration_stmt" on page 22</i>
corba_class	reserved
count	used to determine the size of an array; see <i>"expression" on page 39</i>
dll_class	see <i>"Using the DLL interface" on page 60</i>
do	see <i>"do_while_stmt" on page 33</i> , see <i>"for_stmt" on page 31</i>
else	see <i>"if_stmt" on page 29</i>
elseif	see <i>"if_stmt" on page 29</i>
enum	see <i>"enum_stmt" on page 15</i>
extends	defines the base class; see <i>"class_stmt" on page 25</i>
for	see <i>"for_stmt" on page 31</i>
if	see <i>"if_stmt" on page 29</i>
in	binary operator; see <i>"expression" on page 39</i>
import	see <i>"import_stmt" on page 18</i>
include	see <i>"include_stmt" on page 17</i>
index	is the index value of an string indexed array element; see <i>"Working with arrays" on page 51</i>
int	predefined integer type; see <i>"type_identifier" on page 45</i>
namespace	see <i>"import_stmt" on page 18</i>
null	predefined reference to "no object"; see <i>"Working with reference variables" on page 53</i>
ole_class	see <i>"class_stmt" on page 25</i>
return	leaves a function and returns a value to the caller; see <i>"return_stmt" on page 36</i>
set	see <i>"set_stmt" on page 16</i>

string	predefined string type; see <i>"type_identifier" on page 45</i>
super	predefined variable, available in functions of derived classes, pointing to the members of the base class
syn	see <i>"syn_stmt" on page 14</i>
text_table	see <i>"text_table_stmt" on page 16</i>
this	predefined variable, available in functions of a class; used to pass the reference to the current object to a function etc.
throw	see <i>"throw_stmt" on page 34</i>
to	see <i>"for_stmt" on page 31</i>
try	see <i>"try_catch_stmt" on page 34</i>
variant	see <i>"Definition of OLE interfaces in CBL" on page 57</i>
virtual	reserved
void	predefined type; used if a function returns no value or if a reference to an unknown type should be defined; see <i>"type_identifier" on page 45</i>
while	see <i>"while_stmt" on page 32</i> ; see <i>"do_while_stmt" on page 33</i>
with	see <i>"with_stmt" on page 35</i>

3.3 Advanced topics

- > chapter 3.3.1, "Working with arrays"
- > chapter 3.3.2, "Working with reference variables"
- > chapter 3.3.3, "Using the OLE interface"
- > chapter 3.3.4, "Using the DLL interface"
- > chapter 3.3.5, "Working with the SOAP Interface"
- > chapter 3.3.6, "Import of XSD Files"
- > chapter 3.3.7, "Import of ASN.1-Modules"

3.3.1 Working with arrays

This section summarizes the CBL features regarding arrays.

- Array with "unlimited" size:

```
int array[];
```

- Array with limited size:

```
int array[10];
```

- Integer indexed array (unlimited):

```
string array[];  
array[1]="Hugo";
```

- Initialization of a integer indexed array:

```
string array[]={ "Hans", "Bernd", "Roland" };
```

- String indexed array (unlimited):

```
string array[];  
array["Erich"]="1234";
```

- Initialization of a string indexed array:

```
string strarray[]={ "Erich":"1234", "Bernd":"4321", "Roland":"1111" };
```

- An array can be set to 'empty' using the trivial form of expression list assignment:

```
strarray={};
```

- String indexed arrays are always sorted by increasing alphanumerical index string.

- Getting the size of an array:

```
strarray.count
```

- A string indexed array can also be accessed with integer indices from 0 to count-1.

- Getting the index-string of an string indexed array element:

```
string name=strarray[2].index; /* name becomes "Roland" */
```

- Checking, if there is an array element with the specified string index in a string indexed array:

```
int i=ContainsStrIndex(strarray,"Roland"); /* i becomes 1 */  
int j=ContainsStrIndex(strarray,"Alexandra"); /* j becomes 0 */
```

If the first parameter is not a string indexed array, a runtime error occurs.

- `ArrayRemoveAt` can be used to delete one specific element from an array identified by it's string index or sort order:

```
ArrayRemoveAt(strarray,"Roland");  
ArrayRemoveAt(strarray,1);
```

3.3.2 Working with reference variables

For efficient working with reference variables you must know how initialization of types and objects as well as assignment are handled within the interpreter.

A "type" or class is a template object. Some template objects are predefined (int, string). The definition of a class (see "*class_stmt*" on page 25) causes a template object to be generated:

```
class complex
{ int re;
  int im;
}
```

"complex"

re
im

Therefore you can also initialize the members in a class declaration to define other initial values than the default 0 values:

```
class point
{ int x = 1;
  int y = 9;
}
```

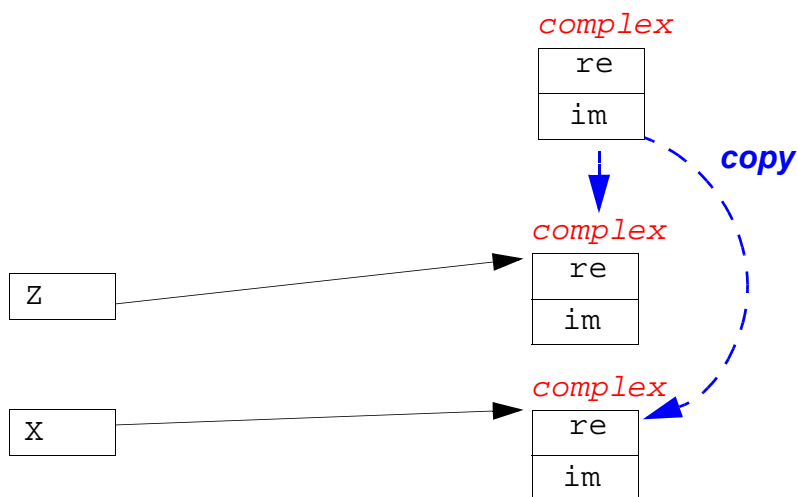
"point"

x 1
y 9

A structured variable (e. g. an array or class variable) is a pointer to its elements.

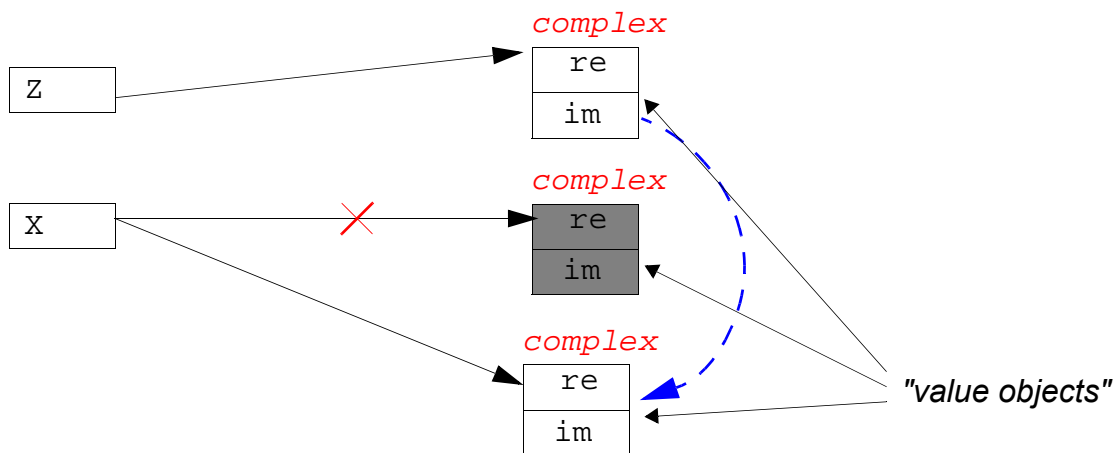
The definition of a variable causes the template object to be "cloned" (copied) and the definition of a reference to this new object to be stored in the variable. Therefore the newly defined variable has the same initial values as the template object for the class:

```
complex Z;
complex X;
point P; // P.x is 1 and P.y is 9
```



When a structured variable is assigned to another structured variable as a whole, the value object of the source variable is cloned and assigned to the destination variable (Btw.: This is not the way the interpreter really works, but this description shows it from the users view):

`X=Z;`

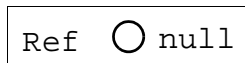


As shown, the previous value object still exists but without any reference to it. Value objects without any reference will be removed by a process called "Garbage Collection".

However, when an expression list is assigned to a structured variable, the assignment occurs elementwise. Therefore, no object is cloned and the memory used by the object remains the same!

Reference variables allow a named data element to be referred multiply. If a reference variable is created, it first points to no value object. This value is called "null".

```
complex @Ref;
```



The reference variable cannot be accessed until it has been assigned a variable.

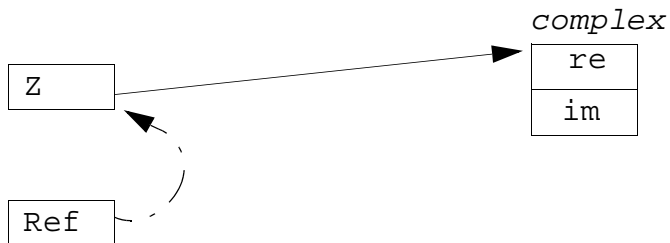
The only thing you can do here, is to test if the reference variable is null:

```
if (Ref == null) ...
```

```
if (Ref != null) ...
```

When you assign a variable to a reference variable which is null, a reference to the right side of the assignment is stored in the reference variable:

```
Ref=Z;
```



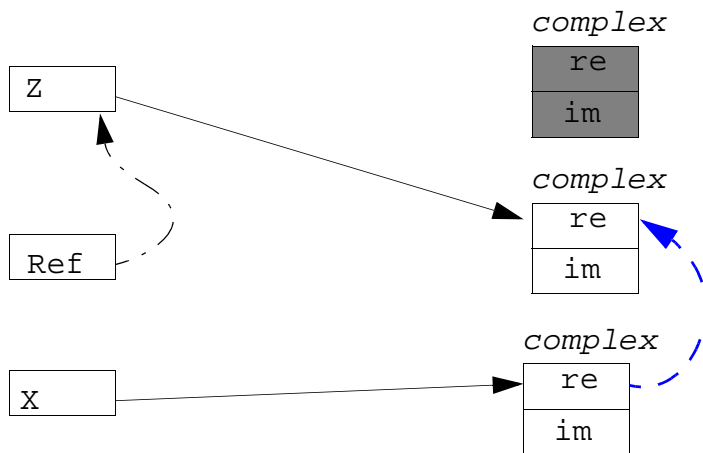
Ref is now a synonym for Z. If Ref is assigned a new value the value of the referred variable changes too:

```
Ref=X;
```

```
Z=X;      // both assignments have the same effect!
```

The CBL Language

Advanced topics



A reference variable can be reset to its original value using `Ref=null`.

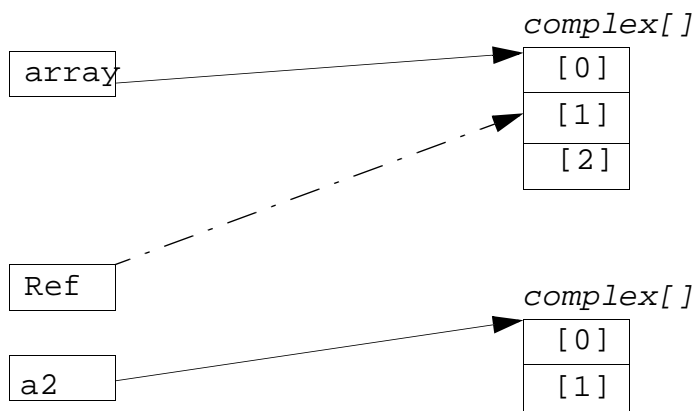
Memory management by garbage collection assures, that a non-null reference is always referring to valid data.

However, you have to be careful when storing references to array elements or class members. If the array variable as a whole is assigned a new value the reference variable will still point to the old array element!

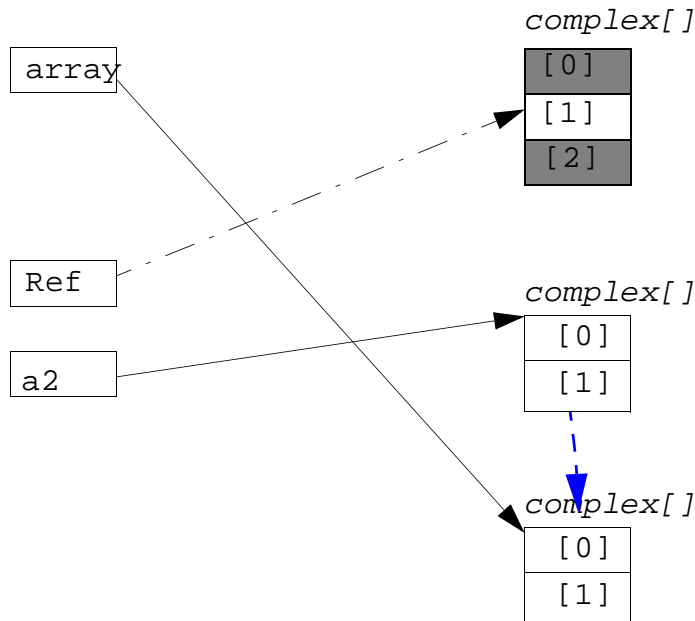
The same happens, if a reference variable points to a class member and the complete object is assigned a new value.

```

complex array[3];
complex @Ref=array[1];
complex a2[2];
  
```



```
array=a2;
```



Now Ref doesn't point to the current array[1] anymore!

The grayed value objects will be deleted at next garbage collection.

3.3.3 Using the OLE interface

From the CBL syntax perspective the definition see *"class_stmt"* on page 25.

With the OLE interface you can access either existing 3rd party programs like MS-Word, Netscape Browser, etc. or self written OLE servers. The following sections show how to program your own OLE server and the ways to access 1st and 3rd party OLE servers.

3.3.3.1 Definition of OLE interfaces in CBL

To be able to access an OLE object from CBL the object must be modeled with the CBL language. The definition for the above OLE object looks like this:

```
ole_class cblFile : "cblsys1.file"
{
    string open(string name, string mode) : "open"
    string readln() : "readln"
    int write(string txt) : "write"
    void close() : "close"
    int eof : "eof"
}
```

The information which is important for the logical link between OLE and CBL is printed bold.

In the current version of the interpreter only the types `int` and `string` are allowed for parameters, return values and properties. Only Call-By-Value is supported.

The function parameters, return values and property types must be consistent between OLE and CBL.

If you want to access 3rd party applications you must try to get the interface from the vendors help files or documentation.

Examples: (works only for German MS Word 7.0)

```
ole_class MS_Word : "Word.Basic"
{
    void FileNew(string type)           : "DateiNeu"
    void Insert(string text)            : "Einfügen"
    void FileSaveAs(string param)       : "DateiSpeichernUnter"
    void FormatTabulator(string pos)    : "FormatTabulator"
}
ole_class Netscape : "Netscape.Network.1"
{
    int Open(string pURL, int iMethod, string pPostData,
              int lPostDataSize, string pPostHeaders) : "Open"
    int GetStatus()                                : "GetStatus"
    void Close()                                   : "Close"
    int BytesReady()                               : "BytesReady"
    int GetContentLength()                         : "GetContentLength"
    string GetContentType()                       : "GetContentType"
    int Read(string pBuffer, int iAmount) : "Read"
}
```

3.3.3.2 Using immediate OLE functions

Some 3rd party OLE servers use very complicated interfaces that cannot be modeled by means of CBL. Therefore the Interpreter offers some low level functions for communication with arbitrary OLE servers.

Example:

```
class Variant
{
    variant obj;
    void CreateObject(string name)
    { _OleCreateObject(obj,name); }
    variant OlePropertyGet(string name,variant i1, variant i2)
```

```

{ variant value;
  _OlePropertyGet(obj,name,value,i1,i2);
  return value; }
void OlePropertySet(string name,variant value,variant i1, variant i2)
{ _OlePropertySet(obj,name,value,i1,i2); }
void OleProcedure(string name,variant p1, variant p2, variant p3)
{ _OleProcedure(obj,name,p1,p2,p3); }
variant OleFunction(string name,variant p1, variant p2, variant p3)
{ variant value;
  _OleFunction(obj,name,value,p1,p2,p3);
  return value; }
}
Variant sheet;
sheet.CreateObject("Excel.Sheet");
Variant app;
app.obj=sheet.OlePropertyGet("Anwendung",empty,empty);
int visible=app.OlePropertyGet("Sichtbar",empty,empty);
app.OlePropertySet("Sichtbar","Wahr",empty,empty);
Variant cell;
cell.obj=app.OlePropertyGet("AktiveZelle",empty,empty);
cell.OlePropertySet("Z1S1Formel", "11", empty,empty);
Variant range;
range.obj = sheet.OlePropertyGet("Bereich","A2",empty);
range.OleProcedure("Auswählen",empty,empty,empty);
cell.obj=app.OlePropertyGet("AktiveZelle",empty,empty);
cell.OlePropertySet("Z1S1Formel", "222",empty,empty);
range.obj=sheet.OlePropertyGet("Bereich","A3",empty);
range.OlePropertySet("Z1S1Formel",
                    "=SUMME(Z(-2)S:Z(-1)S)",empty,empty);

```

The Functions `_OleCreateObject`, `_OlePropertyGet`, `_OlePropertySet`, `_OleFunction` and `_OleProcedure` offer a C-like procedure interface to OLE. In this example these function have been wrapped by a class "Variant" which will be offered in a CBL include file. "variant" is the CBL value type that can either be an int, a string or an OLE object.

`CreateObject(string name)` is called to create an interface object to the OLE server. The object is stored in `obj`. `name` is the name of the OLE class as stored in the Windows registry database (e. g. "Netscape.Network").

`variant OlePropertyGet(string name,variant i1, variant i2)` returns the value of property name of the OLE object `obj`. If the property is an array `i1` and `i2` are used as index values to this array. The result can be a string, int or an OLE object.

`void OlePropertySet(string name,variant value,variant i1, variant i2)` sets the value of property name of OLE object `obj` to `value`. If the property is an array `i1` and `i2` are used as index values to this array.

`OleProcedure(string name,variant p1, variant p2, variant p3)` calls the OLE procedure name. `p1`, `p2` and `p3` are parameters for this procedure call.

`variant OleFunction(string name,variant p1, variant p2, variant p3)` calls the OLE function name. `p1`, `p2` and `p3` are parameters for this procedure call. It returns the result of the function call.

3.3.4 Using the DLL interface

For a definition from the perspective of CBL syntax, see "*class_stmt*" on page 25.

Defining a DLL class means that functions that are implemented in a DLL library can be called up directly from a CBL script. Understandably, this interface is subject to various restrictions. These are:

- The DLL function must be declared to be a WINAPI (due to the parameter transfer sequence)
- The only parameter types allowed are
`int` (corresponds to 32-bit long in C/C++),
`int@` (corresponds to long & in C/C++) and
`string` (converted into `char *` in C/C++). Unfortunately, Unicode strings are not supported.
- Memory blocks can be administered via the new CBL-internal data type `_memblock` and the associated access routines `_MemBlockGetPtr`, `_MemBlockSetPtr`, `_MemBlockAlloc`, `_MemBlockFree`, `_MemBlockGetInt`, `_MemBlockSetInt`, `_MemBlockGetString` and `_MemBlockSetString`. These functions can be encapsulated with the following "wrapper" class:

```
class cblMemblock
{
    _memblock mb;
    int    GetPtr()           { return _MemBlockGetPtr(mb); }
    void    SetPtr(int ptr)   { _MemBlockSetPtr(mb,ptr); }
    void    Alloc(int size)   { _MemBlockAlloc(mb,size); }
    void    Free()            { _MemBlockFree(mb); }
    int     GetInt(int offset, int size)
                                { return _MemBlockGetInt(mb,offset,size); }
    void     SetInt(int offset, int size, int value)
                                { _MemBlockSetInt(mb,offset,size,value); }
    string  GetString(int offset, int size)
                                { return
```

```

_MemBlockGetString(mb,offset,size); }
void SetString(int offset, int size, string s)
{ _MemBlockSetString(mb,offset,size,s); }
}

```

Meaning of the functions:

GetPtr	Returns the address of the memory block as an int value
SetPtr	An int value which represents a memory block address must be included as ptr. Otherwise, unforeseen program reactions can occur (e.g. access violations)
Alloc	Seizes a memory block of the specified size
Free	Frees the current memory block
GetInt	Interprets size bytes from the offset address offset (relative to the memory block start) as an int value and supplies this value as return.
SetInt	Interprets size bytes from the offset address offset (relative to the memory block start) as an int value and sets this to value.
GetString	Interprets size bytes from the offset address offset (relative to the memory block start) as a null-terminated character string and supplies it as a return value.
SetString	Copies maximum size bytes from the character string s to the address offset (relative to the memory block start) and terminates this, where applicable, with '\0'.

- Only void, int or string are allowed as return values for functions.

Example: Calling functions from USER32.DLL:

```

dll_class DLL_User32 : "User32.dll"
{
    int MessageBeep(int x) : "MessageBeep"

    int MessageBox(int hWnd, string lpText, string lpCaption, int uType)
        : "MessageBoxA"
}
DLL_User32 user32;

user32.MessageBeep(1000);
user32.MessageBox(0,"Hello World", "Message", 1);

```

3.3.5 Working with the SOAP Interface

Precondition for this feature is an installation of Microsoft SOAP Toolkit SDK 3.0. In later ComWin version this toolkit will be installed with ComWin.

The interface to WebServices can be read using the import statement (see *"import_stmt"* on page 18).

Example: Import of a WSDL via HTTP

```
import "http://localhost/XAWS/XA.asmx?WSDL" namespace XA;
```

If the WSDL is stored in a local file it can be imported immediately as well:

Example: Import of a WSDL file

```
import "xa.wsdl" namespace XA;
```

If the server requires authentication with user and password the CBL Interpreter intrinsic function `_Import` can be called.:

Example: Import of a WSDL using function `_Import`

```
try
{
    _Import("http://" + ServerIP + "/HiT/" + ws + ".asmx?WSDL", ns, User, Password);
}
catch(SoapException e)
{
    print_SoapException(e);
}
```

As shown in the example above the URL can be constructed at runtime. Exceptions can be caught using the CBL exception handling. The second parameter of the `_Import` function is the namespace.

If a WebService could be imported successfully there will be a class with the name of the WebService and methods for each WebMethod. These types and objects can be visualized using the data inspector of the CBL interpreter.

These imported objects contain predefined properties like `_url`, `_keepconnection` and `_http`. They are used to control the execution of the methods of that WebService.

`_url` is a text string. It tells the CBL interpreter where to execute the WebMethod. Therefore the `_url` must be set in advance to the execution of the first method. Setting `_url` is absolutely necessary if the WSDL was read from a file.

`_keepconnection` tells the CBL interpreter if the HTTP connection should be reset (0) or kept (1) for each call of a WebMethod. Default value of `_keepconnection` is 1.

By means of the string array `_http` additional parameters can be passed to the HTTP connector component (see description of MS-SOAP-Toolkit for details).

Methods that are defined in the WebService can be called like ordinary CBL functions::

Example: Call to a WebMethod:

```
XA::XA xa;  
xa._url="http://localhost/XAWS/XA.asmx";  
string result=xa.SetPhoneState("31562","Online","2003-10-15 12:00");
```

3.3.6 Import of XSD Files

XSD files are XML schema descriptions. These files describe valid syntax of XML files.

Such files can be included by means of the import statement. (see *"import_stmt"* on page 18). Due to the suffix ".xsd" the CBL Interpreter recognizes these files as XML schema files.

Example: Import of a XSD file:

```
import "C:\\localdata\\fe31562\\ECMA\\make-call.xsd" namespace CSTA
```

XSD files may contain references to further XSD files. These will be read and interpreted as well.

This shows an example of a XSD file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 U (http://www.xmlspy.com) by tom miller (self) -->
<xsd:schema targetNamespace="http://www.ecma.ch/standards/ecma-323/csta/ed2"
xmlns:csta="http://www.ecma.ch/standards/ecma-323/csta/ed2" xmlns:xsd="http://www.w3.org/2001/
XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xsd:annotation>
    <xsd:documentation>CSTA-make-call</xsd:documentation>
  </xsd:annotation>
  <xsd:include schemaLocation="device-identifiers.xsd"/>
  <xsd:include schemaLocation="call-connection-identifiers.xsd"/>
  <xsd:include schemaLocation="device-feature-types.xsd"/>
  <xsd:include schemaLocation="extension-types.xsd"/>
  <xsd:include schemaLocation="call-control.xsd"/>
  <xsd:include schemaLocation="media-services.xsd"/>
  <xsd:element name="MakeCall">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="callingDevice" type="csta:DeviceID"/>
        <xsd:element name="calledDirectoryNumber" type="csta:DeviceID"/>
        <xsd:element ref="csta:accountCode" minOccurs="0"/>
        <xsd:element ref="csta:authCode" minOccurs="0"/>
        <xsd:element name="autoOriginate" type="csta:AutoOriginate" default="prompt"
minOccurs="0"/>
        <xsd:element ref="csta:correlatorData" minOccurs="0"/>
        <xsd:element ref="csta:userData" minOccurs="0"/>
        <xsd:element ref="csta:callCharacteristics" minOccurs="0"/>
        <xsd:element ref="csta:mediaCallCharacteristics" minOccurs="0"/>
        <xsd:element name="callingConnectionInfo" type="csta:ConnectionInformation"
minOccurs="0"/>
        <xsd:element ref="csta:subjectOfCall" minOccurs="0"/>
        <xsd:element ref="csta:languagePreferences" minOccurs="0"/>
        <xsd:element ref="csta:extensions" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="MakeCallResponse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="callingDevice" type="csta:ConnectionID"/>

```

```
<xsd:element ref="csta:mediaCallCharacteristics" minOccurs="0"/>
<xsd:element name="initiatedCallInfo" type="csta:ConnectionInformation" minOccurs="0"/>
<xsd:element ref="csta:extensions" minOccurs="0"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

The elements defined in this file lead to definition of corresponding data structures in the symbol table of the CBL interpreter. These types can then be inspected using the data inspector. With the data inspector you can also navigate to the XSD file where the data structure has been defined. This is a very helpful function when exploring complex schemas.

If a XML string is to be created - matching the XML description shown above - an object of this data type can be instanceated like this:

```
CSTA::MakeCall makecall;
```

The properties of this object can be accessed like with any other CBL defined object:

```
makecall.callingDevice="31562";
makecall.callingDevice.typeOfNumber=
    CSTA::DeviceID::typeOfNumber::dialingNumber;
makecall.calledDirectoryNumber="39153";
```

Each class that has been generated by a XSD import has the methods `string ToString()` and `void FromString(string s)`.

Using `ToString()` a XML string will be generated from the current objects content. Default values will be taken into consideration when generating the XML string: only values that differ from the default values will be visible in the string.

```
string s=makecall.ToString();
comwinFrame.ConsolePrint(_BeautifyXML(s));
```

The function `_BeautifyXML` can be used to “beautify” a XML string to enhance its readability.

The object shown above results in this XML string:

```
<MakeCall>
  <callingDevice>31562</callingDevice>
  <calledDirectoryNumber>39153</calledDirectoryNumber>
</MakeCall>
```

The method `FromString` is used for parsing a XML string in the other direcion. The object will be filled with data provided in the XML string:

The CBL Language

Advanced topics

```
CSTA::MakeCall mc;  
mc.FromString(s);
```

The the data can be accessed using ordinary CBL functionality. By this means the CBL interpreter got a validating XML parser.

3.3.7 Import of ASN.1-Modules

By means of the import statement (see *"import_stmt"* on page 18) ASN.1-Modules can be "imported" into a CBL project.

Therefor the desired module or a sum of zip packed modules will be given to the ASN.1 compiler "snacc". This freeware tool was enanced to generate CBL code. "snacc" is called in the background in a way that the generated CBL files are placed into a temporary directory and added to the current CBL project.

The imported class definitions are visible in the data inspector, the generated modules are visible in the module list of the CBL Interpreter.

Unfortunately not all ASN.1 dialects can be interpreted using snacc (version 1.3); ASN.1 had some evolution steps in its past. But the functionality is enough to interpret the complex specifications of the ECMA CSTA III protocol.

The usage of the BER encoding, decoding and printing routines is analoy to the generated C++ code. The advantage of CBL code is that you do not have to care about memory management. This is done automatically by the CBL garbage collector.

This small example shows generation and analysis of ASN.1 coded messages from CSTA "SystemStatusRequest":

Example: Code generation of ASN.1-Code

```
import "csta.zip" namespace CSTA
...
CSTA::ROSEapdus rose_apdu;
CSTA::ROSEapdus rose_res;
...
int bytesDecoded=0;
int bytesEncoded=0;
AsnBuf b;
string os;
void HandleMessage(string hex)
{
    os=" ";
    b.InstallDataFromHex(hex);
    rose_apdu.Init();
    int ok=rose_apdu.BDecPdu(b,bytesDecoded,os);
    if (ok)
    {
        os << endl;
        rose_apdu.Print(os);
        comwinFrame.ConsolePrint("\nReceived ROSE:\n");
        comwinFrame.ConsolePrintStyle(os,cpsFixed);
        with (rose_apdu)
        {
            if (!IsNull(roiv_apdu))
            {
                with (roiv_apdu)
```

```
{
  int send_response=0;
  if (operation_value.value==211)
  {
    // Received SystemStatus-Request --> Send SystemStatusRes
    with (rose_res)
    {
      choiceId=CSTA::ROSEapdus::rors_apduCid;
      rors_apdu=CSTA::RORSapdu::New();
      with (rors_apdu)
      {
        invokeID.value=rose_apdu.roiv_apdu.invokeID.value;
        cSTA::CSTA::RORSapduSeq=CSTA::RORSapduSeq::New();
        with(cSTA::CSTA::RORSapduSeq)
        {
          operation_value.value=211;
          anyDefBy=CSTA::GetAnyByInt(operation_value,2);
          with(anyDefBy)
          {
            choiceId=CSTA::SystemStatusRes::noDataCid;
            noData =AsnNull::New();
          }
        }
      }
    }
    send_response=1;
  }
  elseif (operation_value.value==207) // SystemRegister
  {
    // Received SystemRegister-Request--> Send SystemRegisterResult
    with (rose_res)
    {
      choiceId=CSTA::ROSEapdus::rors_apduCid;
      rors_apdu=CSTA::RORSapdu::New();
      with (rors_apdu)
      {
        invokeID.value=rose_apdu.roiv_apdu.invokeID.value;
        cSTA::CSTA::RORSapduSeq=CSTA::RORSapduSeq::New();
        with(cSTA::CSTA::RORSapduSeq)
        {
          operation_value.value=207;
          //anyDefBy=CSTA::SystemRegisterResult::New();
          anyDefBy=CSTA::GetAnyByInt(operation_value,2);
          with (anyDefBy)
          {
            sysStatRegisterID.octetLen=3;
            sysStatRegisterID.octs={ 1, 2, 3 };
          }
        }
      }
    }
    send_response=1;
  } // SystemRegister
  if (send_response)
  {
    b.InitInWriteMode();
  }
}
```

```

        rose_res.BEncPdu(b,bytesEncoded);
        b.PutByteRvs(bytesEncoded%256);
        b.PutByteRvs(bytesEncoded/256);
        hex=b.ToHex();
        sim.SendRequest(hex,3);
        os="";
        rose_res.Print(os);
        comwinFrame.ConsolePrint("\nSent:\n");
        comwinFrame.ConsolePrintStyle(os,cpsFixed);
    }
}
}
}
return;
}
os="";
b.InstallDataFromHex(hex);
acse_apdu.Init();
ok=acse_apdu.BDecPdu(b,bytesDecoded,os);
if (ok)
{
    os << endl;
    acse_apdu.Print(os);
    comwinFrame.ConsolePrint("\nReceived:\n");
    comwinFrame.ConsolePrintStyle(os,cpsFixed);
    //
    // Waiting for ACSE::AARQ
    //
    // Send ACSE::AARE
    //
    ACSE::ACSE_apdu resapdu;
    resapdu.choiceId=ACSE::ACSE_apdu::aareCid;
    resapdu.aare=ACSE::AARE_apdu::New();
    with (resapdu.aare)
    {
        application_context_name=acse_apdu.aarq.application_context_name;
        result.value=ACSE::Associate_result::accepted;
        result_source_diagnostic=ACSE::Associate_source_diagnostic::New();
        with (result_source_diagnostic)
        {
            acse_service_user=ACSE::Associate_source_diagnosticInt::New();
            acse_service_user.value=ACSE::Associate_source_diagnosticInt::null1;
        }
        user_information=ACSE::Association_information::New();
        with (user_information)
        {
            with(list[0])
            {
                direct_reference=AsnOid::New();
                direct_reference=acse_apdu.aarq.user_information.list[0].direct_reference;
                encoding=ACSE::EncodingCsta::New();
                encoding=acse_apdu.aarq.user_information.list[0].encoding;
            }
        }
    }
}
b.InitInWriteMode();

```

```
    resapdu.BEncPdu(b,bytesEncoded);
    b.PutByteRvs(bytesEncoded%256);
    b.PutByteRvs(bytesEncoded/256);
    hex=b.ToHex();
    sim.SendRequest(hex,3);
    os=" ";
    resapdu.Print(os);
    comwinFrame.ConsolePrint("\nSent:\n");
    comwinFrame.ConsolePrintStyle(os,cpsFixed);
    //
    // Send CSTA/ROSE::SystemStatus
    //
    rose_apdu.Init();
    with (rose_apdu)
    {
        choiceId=CSTA::ROSEapdus::roiv_apduCid;
        roiv_apdu=CSTA::ROIvApdu::New();
        with (roiv_apdu)
        {
            invokeID.value=0;
            operation_value.value=211;
            argument=CSTA::GetAnyByInt(operation_value,1);
            with(argument)
            {
                systemStatus.value=CSTA::SystemStatus1::normal;
            }
        }
    }
    b.InitInWriteMode();
    rose_apdu.BEncPdu(b,bytesEncoded);
    b.PutByteRvs(bytesEncoded%256);
    b.PutByteRvs(bytesEncoded/256);
    hex=b.ToHex();
    sim.SendRequest(hex,3);
    os=" ";
    rose_apdu.Print(os);
    comwinFrame.ConsolePrint("\nSent:\n");
    comwinFrame.ConsolePrintStyle(os,cpsFixed);
    return;
}
}
```


4 CBL - Programmer's Reference

This chapter describes the predefined CBL include files, the contained classes and their usage as well as the variables and functions of the interface to the CBL Interpreter itself.

Advanced topics

- > chapter 4.1, "Interface to the Interpreter"
- > chapter 4.2, "General functions"
- > chapter 4.3, "Interfaces to the HiPath 4000 Expert AccessFrame components"
- > chapter 4.4, "Designing forms"
- > chapter 4.5, "CBL - National Language Support"

4.1 Interface to the Interpreter

This section describes some low level functions/variables of the CBL Interpreter.

Advanced topics

- > chapter 4.1.1, "Command line parameters: Args"
- > chapter 4.1.2, "Interpreter functions"

4.1.1 Command line parameters: Args

Command line parameters can be accessed by a CBL program via the predefined array variable `string [] Args`. This global variable is generated as soon as the CBL program is started.

The last parameter of a command line call to the CBL Interpreter is the name of the desired CBL program.

```
cbllrun.exe par1=value1 par2=value2 par3=value3 test.cbl  
cbllrun.exe /par1 value1 /par2 value2 /par3 value3 test.cbl  
cbllrun.exe -par1 value1 -par2 value2 -par3 value3 test.cbl
```

Then the `Args` array gets the following values:

```
Args["par1"]="value1"
Args["par2"]="value2"
Args["par3"]="value3"
```

4.1.2 Interpreter functions

This sections lists all function that are implemented in the interpreter internally.



Functions starting with underscore _ should only be called directly by the user if absolutely necessary. They are only intended for functions defined in system includes.

<code>string TranslateIn(string)</code>	Translates a string from external representation into application language by using the current text_table entries
<code>string TranslateOut(string)</code>	Translates a string from application language to external representation by using the current text_table entries
<code>int TextHeight(string)</code>	Calculates the height of a text in pixels. To be used for explicit calculation of a text field size on a form.
<code>int TextWidth(string)</code>	Calculates the width of a text in pixels. To be used for explicit calculation of a text field size on a form.
<code>void Wait(int ms)</code>	Suspends the current executed CBL program for ms milliseconds.
<code>int ContainsStrIndex (void @array, string index)</code>	Checks, if there is an array element array [index]. First Parameter array must be an string indexed array, otherwise a runtime error occurs. index is the string used to address the array element, if it exists.
<code>int StartJavaServer (int port)</code>	Starts the built-in Java Interface at the desired port.
<code>void StopJavaServer()</code>	Stops the built-in Java Interface
<code>int GarbageCollection()</code>	Explicitly starts a garbage collection, i. e. memory not used anymore is released. The return value is the number of objects released. GarbageCollection() is also internally called by the interpreter itself. It is a good idea to call GarbageCollection() at times, when the user does not notice this, e. g. after showing a form (see Form::Show() in system include form.cbl) or after updating a form.

<pre>void ArrayRemoveAt(void @array, string index) void ArrayRemoveAt(void @array, int in- dex)</pre>	<p>ArrayRemoveAt can be used to delete one specific element from an array identified by it's string index or sort order.</p>
<pre>void ShowProgress(int _percent, string _comment)</pre>	<p>A window showing progress is displayed. If <code>_percent < 0</code> or <code>> 100</code>, the window is closed, otherwise <code>_percent</code> represents the position of the progress bar. <code>_comment</code> appears in the window's comment line.</p>
<pre>void TracePrint(int _level, string _text)</pre>	<p>Trace messages can be transferred to HiPath 4000 Expert Access's standard trace mechanism with <code>TracePrint</code>.</p>
<pre>void Dispose(void @_obj)</pre>	<p>An object can be explicitly and immediately released with <code>Dispose</code>. Afterwards, it can no longer be accessed as it was removed from the CBL Interpreter's symbol table. References to the server are immediately released, where applicable, in the case of OLE objects.</p>
<pre>int _HttpDownloadFile (string _url, string _filename)</pre>	<p>The function can be used to load a file via HTTP from the specified URL <code>_url</code> (<code>http://...</code>) and save it under <code>_filename</code>.</p>
<pre>int _WaitUntilReleaseWait (int @_context, int _timeout) void _ReleaseWait (int _context, int _return)</pre>	<p><code>_WaitUntilReleaseWait</code> enables the CBL Interpreter to receive events that were produced by an OLE server for the time duration of <code>_timeout</code> milliseconds and process them with the relevant event handling routines. In the meantime, all other events (keyboard inputs, mouse actions) are processed as normal.</p> <p>A value that can be used by <code>_ReleaseWait</code> is returned in <code>_context</code> so that the <code>_WaitUntilReleaseWait</code> function can be prematurely terminated.</p> <p>The return value in the case of a timeout is 0, and in the case of premature termination via <code>_ReleaseWait</code> it is the transferred value of the <code>_return</code> argument.</p>
<pre>int _Defined (void @_this, string _symbol)</pre>	<p><code>_Defined</code> can be used to check if a specific function named <code>_symbol</code> or a specific field is defined for an object.</p>

<code>int _MemBlockGetPtr (_memblock @m);</code>	The _MemBlock functions are needed for the implementation of memory blocks for the DLL interface. see <i>"Using the DLL interface"</i> on page 60
<code>void _MemBlockSetPtr (_memblock @m, int ptr)</code>	
<code>void _MemBlockAlloc (_memblock @m, int si- ze)</code>	
<code>void _MemBlockFree (_memblock @m)</code>	
<code>int _MemBlockGetInt (_memblock @m, int offset, int size)</code>	
<code>void _MemBlockSetInt (_memblock @m, int offset, int size, int value)</code>	
<code>string _MemBlockGetString (_memblock @m, int offset, int size)</code>	
<code>void _MemBlockSetString (_memblock @m, int offset, int size, string s)</code>	
<code>int _MemBlockGetSize (_memblock @m)</code>	
<code>string _BeautifyXML(string s)</code>	This function is used to “beautify” XML strings to enhance it’s readabilty e.g. for printouts. Linefeeds and indents are inserted into the string. Its use is shown in section 3.3.6 on page 64.
<code>_CorbaObjectBind</code>	Not used at present.
<code>_Delete</code>	Delete a specific object.
<code>_FormCheckComplete</code>	Used by form.CheckComplete.
<code>_FormCheckControl</code>	Used by form.CheckControl
<code>_FormClose</code>	Used by form.Close.
<code>_FormGet</code>	Used by form.Get.

<code>_FormShow</code>	Used by <code>form.Show</code>
<code>_FormStoreSizePos</code>	Used by <code>form.StoreSizePos</code>
<code>_FormResetModifiedControl</code>	Used by <code>form.ResetModifiedControl</code>
<code>_FormRetrieveSizePos</code>	Used by <code>form.RetrieveSizePos</code>
<code>_FormUpdate</code>	Used by <code>form.Update</code>
<code>_Import</code>	This function implements the dynamical pendant for the import statement. (see " <i>import_stmt</i> " on page 18). It's usage is shown in section 3.3.5 on page 62.
<code>void _Interrupt()</code>	This function can set a "persistent" stop in CBL source code. If the CBL Interpreter is in "Debug" processing mode (to be set via the Option menu), then processing is interrupted after <code>_Interrupt</code> as if a stop were set there. If the CBL Interpreter is not in "Debug" mode, the <code>_Interrupt</code> calls are simply ignored.
<code>_OleCreateObject</code>	Used by <code>Variant.CreateObject</code>
<code>_OleFunction</code>	Used by <code>Variant.OleFunction</code>
<code>_OleProcedure</code>	Used by <code>Variant.OleProcedure</code>
<code>_OlePropertyGet</code>	Used by <code>Variant.OlePropertyGet</code>
<code>_OlePropertySet</code>	Used by <code>Variant.OlePropertySet</code>
<code>_RemoveLast</code>	Removes last element of an array. Not yet used.
<code>_RunModule</code>	Includes a cbl module at runtime. Not yet used.
<code>_SendMessage</code>	Sends a message string via TCP to an IP/port. Not yet used.
<code>_StartModal</code>	Used by <code>form.StartModal</code> .

4.2 General functions

General functions can be included from the CBL include file sysfunc.cbl.

"General functions" covers functions for string handling, operating system interface, file handling, accessing the Windows registry database and some more.

Advanced topics

- > chapter 4.2.1, "cbl_sysfunc_ (ole_class)"
- > chapter 4.2.2, "cbl_chain (ole_class)"
- > chapter 4.2.3, "cblFile (ole_class)"
- > chapter 4.2.4, "cblIniFile (ole_class)"
- > chapter 4.2.5, "cblZipFile (ole_class)"
- > chapter 4.2.6, "cbl_dialogs (ole_class)"
- > chapter 4.2.7, "cblXmlFile (ole_class)"

4.2.1 cbl_sysfunc_ (ole_class)

The OLE class "cblsys1.sysfunc" is implemented in the DLL cblsys1.dll.

```
ole_class cbl_sysfunc_ : "cblsys1.sysfunc"
{
    // string handling
    int strlen(string s) ⇨
    string substr(string s,int start, int end) ⇨
    int char(string s, int index)⇨
    string trim(string s)⇨
    string ltrim(string s)⇨
    string rtrim(string s)⇨
    string lfill(string s, int count, string fill)⇨
    string stripquotes(string s)⇨
    int strfind(string s1, string pattern, int count=1)⇨
    int match_regex(string s, string regex, int index, string @match)⇨
    string int2hex(int value,int digits) ⇨
    int hex2int(string value)⇨
    string [] string_lines(string _in)⇨
    string [] string_token(string _in, string _sep)⇨
    string toupper(string s)⇨
    string tolower(string s)⇨
    string replace(string s, string old, string new)⇨
    string [] sort_grid(string [] _in, int col)⇨
}
```

```
string format(string _in, string _sep, string _format)⇒

// encryption / decryption
string encrypt(string s, string p="")⇒
string decrypt(string s, string p="")⇒

// operating system related functions/properties
string system(string cmd)⇒
int clock()⇒
string currentDateTime ⇒
int ShellExecute(string operation, string file,
                 string parameters="", string directory="")⇒
void MessageBeep(int sound=-1)⇒

// procedures for AMO result handling
string [] prepare_regen_results(string _in)⇒
string [] param_ivals(string _in)⇒
string [] parse_amo_result(string _in)⇒
string [] parse_amo_errors(string _in)⇒
cblinterval [] param_intervals(string _in)⇒
//
// Registry access
//
int RegistryExistsKey(string RootKey, string Key)⇒
string RegistryReadString(string RootKey, string Key, string item)⇒
int RegistryReadInteger(string RootKey, string Key, string item,
                        int Default=0)⇒
void RegistryWriteString(string RootKey, string Key, string item,
                        string value)⇒
void RegistryWriteInteger(string RootKey, string Key,
                        string item, int value)⇒
string [] RegistryGetKeyNames(string RootKey, string Key )⇒

//
// Misc
//
string ExtractFilePath(string file)⇒
string CreateTempDir() ⇒
string CreateTempFile(string dir) ⇒
void DeleteTempDir(string dir)⇒
int file_compare(string file1, string file2, int size=0)⇒
string [] FindFile(string pattern)⇒
void ForceDirectories(string dir)⇒
string FindConnectProfile(string uw7ipaddr)
```

```
string  FileDate(string file)⇒  
int     SetFileDate(string file, string OsiDateTime)⇒  
cblUrl  ParseUrl(string url)⇒  
}
```

Predefined object for this class:

cbl_sysfunc_ **syslib**

syn **syn**=syslib

4.2.1.1 String manipulation

String length

```
int strlen(string s)
```

`strlen()` returns the number of characters of `s`.

Substring

```
string substr(string s, int start, int len)
```

`substr()` returns a string which is a part of `s`. It begins at the position of `start` and has the length `len`. `start` is 0-based.

ASCII code of a character

```
int char(string s, int index)
```

`char()` returns the ASCII code of the character in the string `s` at the position `index`, which is 0-based.

Removing blanks (spaces)

```
string trim(string s)
```

`trim()` returns the string `s` without any spaces and control characters.

Removing blanks (spaces) from the left

```
string ltrim(string s)
```

`ltrim()` returns the string `s` where all spaces and control characters on the left have been removed.

Removing blanks (spaces) from the right

```
string rtrim(string s)
```


`rtrim()` returns the string `s` where all spaces and control characters on the right have been removed..

Fill a string from the left

```
string lfill(string s, int count, string fill)
```

`lfill()` returns a new string, where string `fill` has been inserted `count` times before string `s`.

Strip quotes off a string

```
string stripquotes(string s)
```

`stripquotes()` returns string `s`, where in the first step trailing blanks and then one Double-Quote (") at the beginning and at the end of `s` has been removed, if there was one.

Search a pattern in a string

```
int strfind(string s1, string pattern, int count=1)
```

`strfind()` returns an integer value of the position of the first or if specified the `count`-th occurrence of `pattern` in `s1`. -1 is returned, if `pattern` is not found.

Turning decimal into hexadecimal

```
string int2hex(int value, int digits)
```

`int2hex()` transforms an integer to a hexadecimal. `value` is the integer number and `digits` is the length of the hexadecimal number the function should return. If `digits` is less than the positions needed, it will be ignored.

Check string against a "regular expression"

```
int match_regex(string s, string regex, int index, string @match)
```

The character string `s` is checked against the regular expression `regex`. The regular expressions supported by the Unix command "egrep" are accepted as `regex`.

A regular expression is nothing else than a search pattern for finding matching patterns in an input string.

Overview of most common regular expressions:

<code>^</code>	Searches the pattern at the beginning of a line.	<code>"^beginnt hier"</code>
<code>\$</code>	Searches the pattern at the end of a line.	<code>"endet hier\$"</code>

CBL - Programmer's Reference

General functions

*	Any number of the preceding character; in the example all lines are searched that contain the word "schon" with any number of preceding spaces:	" *schon"
.	Exactly one character.	".och"
[]	Exactly one of the enclosed characters.	"[Dd]och"
[a-z]	One of the characters matching a range.	"[A-X]och"
[^]	none of the enclosed characters	"[^ln]och"
\	Invalidates the special meaning of the succeeding characters.	"x\\$\>y"
\<	Search pattern at the beginning of a word	"\<doch"
\>	Search pattern at the end of a word.	"ung\>"
\(..\)	Remember enclosed pattern; this can be accessed later using \1. Up to nine patterns can be stored in this way.	"\ (aff\) ig \ lenstark"
x\{m\}	m-timed occurrence of character x.	"s\{3\}"
x\{m,n\}	At least m-, at most n-timed occurrence of character x	"+\{5,8\}"

`match_regex` supplies the number of hits as the return value

The contents of the index-ed hits are returned in `match`.

Example:

```
string s="is invisible and smells like hare";
string found;

int cnt=sys.match_regex(s,"and",1,found);
//-> cnt=1, found="and"

cnt=sys.match_regex(s,"s.",1,found);
//-> cnt=3, found="st"

cnt=sys.match_regex(s,"s.",2,found);
//-> cnt=3, found="si"

cnt=sys.match_regex(s,"rabbit",1,found);
//-> cnt=0, found=""
```

Turning decimal into hexadecimal

```
string int2hex(int value, int digits)
```

`int2hex()` converts an integer to a hexadecimal number. `Value` is an integer. `Digits` indicates the length of the hexadecimal value to be returned by the function. If the value for `digits` is less than the number of required positions it is ignored.

Turning hexadecimal into decimal

```
int hex2int(string value)
```

`hex2int()` turns the hexadecimal number of the string `value` into an integer, that the function will return.

Split a string into lines

```
string [] string_lines(string _in)
```

`string_lines` splits a string that contains linefeed characters (`'\n'`) into separate lines and returns an array of these lines.

Split a string into tokens

```
string [] string_token(string _in, string _sep)
```

`string_token` returns an array of substrings of `_in` which are separated by the separator `string _sep`. Example: `string_token("A.B.C", ".")` returns { "A", "B", "C" }.

Convert string into all uppercase

```
string toupper(string s)
```

`toupper` converts all lowercase letters in string `s` to uppercase and returns the resulting string.

Convert string into all lowercase

```
string tolower(string s)
```

`tolower` converts all uppercase letters in string `s` to lowercase and returns the resulting string.

Replace a pattern within a string

```
string replace(string s, string old, string new)
```

`replace` replaces all occurrences of string `old` in string `s` with string the contents of string `new`.

Sorting a separated string array by a certain column

```
string [] sort_grid(string [] _in, int col)
```

CBL - Programmer's Reference

General functions

`sort_grid` sorts a string array, whose entries are intended as the contents of a multi-column `ctListbox` control (see [ctListbox](#)), by a specific column defined by `col`. The first column has the index 0.

Format a text string

```
string format(string _in, string _sep, string _format)
```

`format` can be used to format the contents of a string, e.g. for print-editing. The input string `_in` is first split into one or more components, as defined by the separator `_sep`, and then combined again, as defined by the format string `_format`, before being output. The individual output fields in the formatted string are output as defined by `%s` and the corresponding format extensions. The format extensions correspond to the `printf` command in the C Reference.

Examples:

```
_in="AAA,BB,CCCC"
```

```
_sep=" , "
```

```
_format="%s-%s-%s"      --> "AAA-BB-CCCC"
```

```
_format="%5s-%5s-%5s"   --> "  AAA-   BB-  CCCC"
```

```
_format="%-5s+%-5s+%-5s" --> "AAA  +BB   +CCCC "
```

Simple encryption/decryption functions

```
string encrypt(string s, string p="")
```

```
string decrypt(string s, string p="")
```

`encrypt` returns string `s` in an encrypted version. It can be decrypted using `decrypt`. The encrypted string consists of printable characters. Its size is about twice as big as the decrypted string. Optionally you may pass a password `p` for the encryption. This is necessary for a successful decryption then.

These functions can be used to store/retrieve data which should not be stored in readable format (e. g. passwords).

4.2.1.2 Operating system oriented functions

Execute command line commands

```
string system(string cmd)
```

With this function it is possible to pass commands directly to the operating system. If the command does not succeed, the function will return the error message.

Example (starting the *Netscape-Communicator* in *WindowsNT/Windows95*):

```
sys. system("start C:/Programme/NETSCAPE/Communicator/Program/" +  
            "netscape.exe -browser -k -h "+  
            "http://hpserv99.mch.pn.siemens.de:8080/~user");
```

Execution time

```
int clock()
```

The function `clock()` returns a time value in milliseconds (ms). The absolute value is meaningless for the caller. However, it is possible to measure the time between two occurrences. If the execution time is not available, or the value cannot be interpreted, the function will return -1.

Current date time

```
string currentDateTime
```

The property `currentDateTime` returns the current date time in the CBL serialization format. This means that `currentDateTime` can be assigned to an object of type `cblDateTime` to be able to access year, month, day, hour, minute and sec

Open file with the linked program

```
int ShellExecute(string operation, string file,  
                 string parameters="", string directory="")
```

`ShellExecute` (see also Win32 API function "`ShellExecute`") can be used to open a file with the specially defined `operation` method. Optional callup `parameters` and a `directory` can also be specified.

Examples:

```
sys.ShellExecute("open", "textfile.txt"); // opens notepad  
sys.ShellExecute("open", "file.doc");     // opens Word  
sys.ShellExecute("print", "file.doc");    // prints via Word
```

Emit beep

```
void MessageBeep(int sound=-1)
```

A beep or a system sound can be output with `MessageBeep`.

4.2.1.3 Public routines handling AMO-results

Prepare regeneration results for further processing

```
string [] prepare_regen_results(string _in)
```

The `prepare_regen_results()` function creates a 0-based array of all the regen results. Every line is stored in a separate string. The string `_in` is the reply, sent by HICOM, on a REGENERATE-command.

Split linked values of a parameter of an AMO

```
string [] param_ivals(string _in)
```

The `param_ivals()` function creates a 0-based array of the linked values of a parameter of an AMO. The list of the linked values separated with an ampersand (&) or dash (-) has to be the string `_in`.

Example 1:

```
Array=param_ivals("1 & 5 & 6 & 13");
```

The result of this call would be:

```
Array: [0]:"1", [1]:"5", [2]:"6", [3]:"13"
```

Example 2:

```
Array=param_ivals("1 && 5 & 13");
```

The result of this call would be:

```
Array: [0]:"1", [1]:"2", [2]:"3", [3]:"4", [4]:"5", [5]:"13"
```

This is especially useful for displaying all the values in a range in a combobox or listbox.

Example 3:

Variant 2 (values separated by dash) is especially useful for splitting LAGE parameters into its components:

```
param_ivals("1-1-7-13");
```

The result of this call would be:

```
Array: [0]:"1", [1]:"1", [2]:"7", [3]:"13"
```

Splitting a parameter into intervals

```
class cblinterval
{
    int lower;
    int upper;
}
```

```
interval [] param_intervals(string _in)
```

The `param_intervals()` function splits a parameter list into intervals.

Example:

```
sys.param_intervals("1 & 10 && 20 & 30");
```

The result is:

```
interval[0].lower = 1      interval[0].upper = 1
interval[1].lower = 10     interval[1].upper = 20
interval[2].lower = 30     interval[2].upper = 30
```

Results of a HICOM command

```
string [] parse_amo_result(string _in)
```

The `parse_amo_result()` function creates a 0-based array of the results of any HICOM command. Every line in a separate string. The string `_in` is the reply, sent by HICOM, on a HICOM command.

Error numbers

```
string [] parse_amo_errors(string _in)
```

The `parse_amo_errors()` function creates a 0-based array with the error numbers of the results of a HICOM command. All M-, S-, H- and F-messages are stored in the array. The string `_in` is the reply, sent by HICOM, on a HICOM command.

4.2.1.4 Access to the Windows registry database

Check for the existence of an registry key

```
int RegistryExistsKey(string RootKey, string Key)
```

`RegistryExistsKey` checks if the desired `Key` exists under the given `RootKey`. The `RootKey` can take the values "HKEY_LOCAL_MACHINE", "HKEY_CURRENT_USER" etc. The function returns 1 if the key exists, else 0.

Read a string value from the registry

```
string RegistryReadString(string RootKey, string Key, string item)
```

`RegistryReadString` returns the desired string value of the desired `item` at the given `Key` and `RootKey`. The `RootKey` can take the values "HKEY_LOCAL_MACHINE", "HKEY_CURRENT_USER" etc. If the value could not be read the function returns "".

Read an integer value from the registry

```
int    RegistryReadInteger(string RootKey, string Key, string item,
                           int Default=0)
```

This function is identical to `RegistryReadString`, except for the fact that an integer value is read in this case.

Write a string to the registry

```
void    RegistryWriteString(string RootKey, string Key, string item,
                           string value)
```

The function `RegistryWriteString` writes the string value `value` to the specified position in the registry, as defined by `RootKey`, `Key` and `item`. If no such entry exists, a new entry is created with the given value.

Write an integer value to the registry

```
void    RegistryWriteInteger(string RootKey, string Key,
                           string item, int value)
```

The function `RegistryWriteInteger` writes the integer value `value` to the specified position in the registry, as defined by `RootKey`, `Key` and `item`. If no such entry exists, a new entry is created with the given value.

Get subkeys from the registry

```
string [] RegistryGetKeyNames(string RootKey, string Key )
```

`RegistryGetKeyNames` returns the names of all subkeys of the specified `Key` from the registry in the form of a string array.

4.2.1.5 Miscellaneous Functions

Extract File Path

```
string ExtractFilePath(string path)
```

`ExtractFilePath` extracts the file path of the given file path.

Create a temporary directory

```
string CreateTempDir()
```


`CreateTempDir` creates a temporary directory. The path of the directory is returned. The directory is located beneath Window's temporary directory, usually `C : /TEMP`. The name of the temporary directory starts with "CBL".

Create a temporary file

```
string CreateTempFile(string dir)
```

`CreateTempFile` creates a temporary file at the given directory `dir`. The file name starts with "CBL". The name of the file is returned.

Delete a temporary directory

```
void DeleteTempDir(string dir)
```

`DeleteTempDir` tries to delete all files and subdirectories of directory `dir`. The value of `dir` must start with "CBL", this means it should have been created using `CreateTempDir`.

Compare files

```
int file_compare(string file1, string file2, int size=0)
```

`file_compare` compares the given files, opened in binary mode. If `file1` could not be opened it returns -1; if `file2` could not be opened it returns -2.

`size` defines the minimum number of bytes to be compared. If `size` equals 0 (default) the files are compared completely.

`file_compare` returns 0 for equal, 1 for different files.

Find files

```
string [] FindFile(string pattern)
```

`FindFile` returns all file names that match the specified `pattern` (which can be defined using the usual wildcards * and ? allowed under Windows).

Create directory path

```
void ForceDirectories(string dir)
```

`ForceDirectories` creates all directories from the specified path `dir` if the corresponding directories do not exist.

Read file date

```
string FileDate(string file)
```

`FileDate` returns the date of the specified `file` in OSI format, i.e.: `yyyy-mm-dd hh:mm:ss`.

Set file date

```
int      SetFileDate(string file, string OsiDateTime)
```

`SetFileDate` sets the date for the specified file to the date given in `OsiDateTime`, which is defined in OSI format (yyyy-mm-dd hh:mm:ss).

Parse URL

```
cblUrl  ParseUrl(string url)
```

`ParseUrl` parses (i.e. splits) the `url` that is passed to it as an argument into its respective components:

Type	Field	Meaning
string	proto	Protocol (e.g. "http")
string	user	User name
string	pass	Password
string	host	Name or IP address of host
string	port	Port number
string	path	Path specification
string	filename	File name
string []	params	Array of parameters, indexed by parameter name

Example:

```
cblUrl loc=ParseUrl(
"http://hpserv99.mch4.siemens.de:8080/ComWin/
download.html?par1=value1&par2=value2");
-->
loc.proto="http"
loc.user=""
loc.pass=""
loc.host="hpserv99.mch.pn.siemens.de"
loc.port="8080"
loc.path="/ComWin"
loc.filename="download.html"
loc.params["par1"]="value1"
loc.params["par2"]="value2"
```

4.2.2 cbl_chain (ole_class)

The OLE class "cblsys1.chain" is implemented in the DLL cblsys1.dll.

This class encapsulates functions for handling of concatenated AMO parameter values:

```
ole_class cbl_chain: "cblsys1.chain"
{
    void create(string s) ➡
    void remove_int_from_chain(int i) ➡
    void remove_chain_from_chain(string s) ➡
    string chain_to_string() ➡
}
```

Predefined object for this class:

none

Creating a chain object from a chain string of ints

```
void create(string s)
```

The first step in working with chains of int values is converting this string into a chain object. This action converts the chain into a internal format.

Example:

```
cbl_chain mychain;
mychain.create("1&&5&20&&40");
```

Removing a single int value from a chain object

```
void remove_int_from_chain(int i)
```

Removes an int value from the internal representation of the chain object. To see the resulting chain, you have to convert the chain object back into a string (see chain_to_string() below). Note that the actual parameter may be of type int or of type string due to CBL's type compatibility rules.

Example:

```
mychain.remove_int_from_chain(2);
mychain.remove_int_from_chain("4");
```

Removing a chain of int values from a chain object

```
void remove_chain_from_chain(string s)
```

Removes all values in int chain s from the internal representation of the chain object. To see the resulting chain, you have to convert the chain object back into a string (see chain_to_string() below).

Example:

```
mychain.void remove_chain_from_chain("30&32&36&&38");
```

Convert chain object into string

```
string chain_to_string()
```

Converts the internal representation of a chain object back into a string.

Example:

```
string s;  
s = mychain.chain_to_string();
```

After executing all commands in the examples above the result in s is:

```
"1&3&5&20&&29&31&33&&35&39&&40"
```

4.2.3 cblFile (ole_class)

The OLE class "cblsys1.file" is implemented in the DLL cblsys1.dll.

The class encapsulates the access to text files on the hard disk/network etc.

```
ole_class cblFile : "cblsys1.file"  
{  
    string open(string name, string mode) ➡  
    string readln() ➡  
    string read(int size=0) ➡  
    int write(string txt) ➡  
    void close() ➡  
    int eof ➡  
}
```

Opening a file

```
string open(string name, string mode)
```

The function open() opens a file. Here, name is the name of the file and mode determines how the file will be opened. Legal mode values are:

"r" open a file for reading (this file must exist)

"w" create a text file for writing (any already existing file with this name will be overwritten)

"a" open a text file with mode append (new data will be appended to the end of the file).

If the function fails, an error message will be returned, otherwise an empty string will be returned.

Example:

```
cblFile f;  
f.open ("sysfunc.cbl", "r");
```

Reading a string

```
string readln()  
string read(int size=0)
```

The function `readln()` reads a line from a file opened in read mode by `open()`. The returned string are the data the function reads.

The function `read` reads a given number (`size`) of characters from the file. If `size` equals 0 (default) the file is read until the end of file is reached.

Writing a string

```
int write(string txt)
```

The `write()` function write strings to a file that was previously opened (and created) for writing.

Closing a file

```
void close()
```

The `close()` function closes a file that was opened by the `open()` function.

End of a file

```
int eof
```

If the end of a file is encountered the `eof` property returns true (1), otherwise it returns 0.

Example:

```
cblFile f;  
string line;  
f.open("readme.txt", "r");  
line = f.readln();  
while (!f.eof) {  
    //handle line read:  
    ...  
    line = f.readln();  
}  
f.close();
```

The block after the *while_stmt* is executed until the end of the file is reached.

4.2.4 cblIniFile (ole_class)

The OLE class "cblsys1.cblIniFile" is implemented in the cblsys1.dll.

It encapsulates functions for convenient accessing .ini files as well as the Windows registry database.

```
const int ftIni=0;
const int ftRegIni=1;

ole_class cblIniFile : "cblsys1.cblIniFile"
{
    string Open(string name, int type)
    void    Close()
    string ReadString(string _section,string _ident, string _default)
    int     ReadInteger(string _section,string _ident, int _default)
    void    WriteString(string _section,string _ident, string _value)
    void    WriteInteger(string _section,string _ident, int _value)
}
```

Open opens an ini file or a section in the registry below HKEY_CURRENT_USER depending on the value of type. If type==ftIni an ordinary file is opened, else the registry database. If the file is not yet existing it will be generated. If no path is qualified for the file it will be located in the Windows directory.

Close must be used to finish the ini file ore registry database.

ReadString/ReadInteger returns the value of _ident from _section or _default if _ident was not found.

WriteString/WriteInteger stores _value to the identifier _ident in section _section. If the section or identifier is not yet existing it will be generated.

4.2.5 **cblZipFile (ole_class)**

The OLE class "cblsys1.cblZipFile" is implemented in the cblsys1.dll.

It encapsulates functions for working with zipped files.

```
ole_class cblZipFile : "cblsys1.zipfile"
{
    int open(string zipfile)

    string [] directory()

    void      unzip(string destination_dir,
                    string pattern="*.*",
                    int mode=0)

    void      close()
}
```

`open` opens a zip file. It returns 1 if the zipfile could be opened, else it returns 0.

`close` closes the zip file.

`directory` returns the files and paths that are contained within the zip file.

`unzip` unpacks the files matching the given `pattern` into the `destination_dir`.
If `mode` equals 1 a popup window shows the current unpacking progress.

4.2.6 **cbl_dialogs (ole_class)**

The OLE class "cblsys1.dialogs" is implemented in the cblsys1.dll.

It encapsulates functions for working with Windows' standard dialogs.

```
// message-type
enum { mtWarning, mtError, mtInformation, mtConfirmation, mtCustom };
// message-buttons
set  { mbYes, mbNo, mbOK, mbCancel, mbAbort, mbRetry, mbIgnore,
      mbAll, mbNoToAll, mbYesToAll, mbHelp };
// message-result
enum { mrNone, mrOk, mrCancel, mrAbort, mrRetry, mrIgnore, mrYes, mrNo,
      mrAll, mrNoToAll, mrYesToAll };

ole_class cbl_dialogs: "cblsys1.dialogs"
{
    int OpenDialog()
    int SaveDialog()
    string FileName
}
```

```
string Directory
string Filter

int MessageDialog(string message, int type, int buttons)
}
```

`OpenDialog()/SaveDialog()` opens a standard file open/save dialog. It returns 1 if the user pressed the OK button; otherwise, 0. The property `Directory` is used as starting directory for the dialog, the property `FileName` is the default value for the file name within the dialog. If `OpenDialog/SaveDialog` returns 1 the property `FileName` is set to the desired file name. It contains the absolute file path. Filter can be used to set the file filter for the dialogs. The format for Filter is shown in this example:

```
Filter="All files (*.*)|*.*||Text files (*.txt)|*.txt";
```

`MessageDialog` opens a standard message dialog. `message` contains the message to be displayed, `type` defines the type (and symbol) of the message dialog. The displayed buttons are defined by a sum of the message button values. `MessageDialog` returns the button which has been pressed.

Example:

```
cbl_dialogs dlg;
if (dlg.MessageDialog("Are you sure ?",
                    mtConfirmation,
                    mbYes+mbNo) == mrYes)
{ /*...*/ }
```

4.2.7 cblXmlFile (ole_class)

The OLE class "cblsys1.xmlfile" is implemented in the DLL `cblsys1.dll`.

This class implements a simple, non-validating XML parser.

```
class cblXmlRecord
{
    string      name;
    string      [] params;

    cblXmlRecord [] items;
    string      [] contents;
}

ole_class cblXmlFile : "cblsys1.xmlfile"
{
    string      header;
    cblXmlRecord body;
```



```
string open(string name)
int    read()
void   write()
void   close()
}
```

An XML file can be opened with `open`.

The file is read in with `read` and the XML file header is saved in `header`. The recursive data structure `body` contains the complete contents of the XML file.

The current contents of `body` are written back to the XML file with `write`.

The XML file is closed with `close`.

A `cblXmlRecord` contains the name of the XML section, the value of the parameter defined in the current section in an appropriate string-indexed string array `params`, the subordinate XML sections `items` and all other text `contents` that are located between the XML sections above and below the subordinate XML sections.

4.3 Interfaces to the HiPath 4000 Expert AccessFrame components

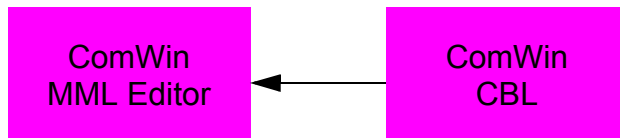
The OLE classes described below are defined in the CBL include file comwin.cbl.

Advanced topics

- > chapter 4.3.1, "Comedit (ole_class)"
- > chapter 4.3.2, "ComwinControl (ole_class)"
- > chapter 4.3.3, "(mode|=2: application handles wrong password, open returns immediately; mode|=4: deactivate autologon.)ComwinFrame (ole_class)"
- > chapter 4.3.4, "ComwinCBL (ole_class)"
- > chapter 4.3.5, "ComWinFT (ole_class)"
- > chapter 4.3.6, "ComWinPPP (ole_class)"
- > chapter 4.4, "Designing forms"
- > chapter 4.3.8, "ComWinAccessSession (ole_class)"
- > chapter 4.3.9, "ComWinAccessFileman (ole_class)"
- > chapter 4.3.10, "ComWinAccessDatabase (ole_class)"
- > chapter 4.3.11, "ComWinAccessQuery (ole_class)"
- > chapter 4.3.12, "ComWinFM (ole_class)"

4.3.1 Comedit (ole_class)

The OLE class "Comedit.MMLCommand" is implemented in the program HiPath 4000 Expert Access-MML Editor.



This class encapsulates the interface from a CBL program to HiPath 4000 Expert Access-MML_Editor. It contains functions for handling of MML command strings and parsing of AMO output strings.

```

ole_class Comedit : "ComWin.MML_Command"
{
  string getParam(string param, int language=0) ⇨
  int     setParam(string param, string value, int language=0) ⇨
  string getParamName(int index, int language=0) ⇨

```

```
string [] getParamNames() ⇨
string getParamDesc(string param, int language=0) ⇨
int     getParamProps(string param) ⇨
string getValue(string param, int index, int language=0) ⇨
string getValueDesc(string param, int index, int language=0) ⇨
string getParamValidation(string param, int language=0) ⇨
int     getParamIndex(string param) ⇨
string command ⇨
int     Active ⇨
int     ParamCount ⇨
string HicomVariant ⇨
int     HicomLanguage ⇨
int     ApplicationLanguage ⇨
string HicomCommand ⇨
string Noun ⇨
string Verb ⇨
string [] GetHicomVariants() ⇨
int     KeywordOriented ⇨
string DestinationProcessor ⇨
int     Error ⇨
string ErrorMessage ⇨
string ErrorParam ⇨
string ErrorParamIdx ⇨
string ErrorValue ⇨
int     Check() ⇨
int     CheckParam(string param) ⇨
int     ExistsParam(string param) ⇨
int     ShowParamDlg(string param) ⇨
string ShowCommandDlg(string command) ⇨
void     ShowCommand(int ConnectionId,
                    String ConnectionName,
                    String MMLCommand,
                    String MMLVariant,
                    int     MMLLanguage,
                    String ReplyObject,
                    String ReplyMethod) ⇨

// AMO output mask parser interface
string AmoResult ⇨
int     AmoResultPos ⇨
string ScanMask(string maskname, int wrparaound=1, int skip=0) ⇨
int     FindMask(string maskname) ⇨
string GetNextMask(string maskname) ⇨
int     SkipLinesUntil(string pattern) ⇨
```

CBL - Programmer's Reference

Interfaces to the HiPath 4000 Expert AccessFrame components

```
int      SkipLines(int cnt) ➡
string  FieldByName(string field) ➡
string  FieldByIndex(int index) ➡
string  FieldName(int index) ➡
int      FieldCount ➡
amoParseMaskRecord [] ParseAmoResult() ➡

// Zusätzliche Funktionen
void     ShowHelp(string url)

// File-oriented check function
int      CheckFileUntilError (string inFile,
                               string outFile,
                               int StartPos,
                               int MaxLines,
                               int Action) ➡

int      ErrorLine
int      FilePos

void CopyFrom(Comedit @src) ➡
}
```

Predefined object for this class

Comedit comedit;

Define the AMO command

string command

The `command` property defines the AMO-command, which is necessary for the other functions of the class.

Example:

```
Comedit comedit;
comedit.command = "EINR-SCSU:2100;";
tlgnu = comedit.getParam("TLGNU");
```

Noun and verb of the current AMO command

string Noun

string Verb

The noun and verb of the current `command` can be read and set via the properties `Noun` and `Verb`.

Initialize to work with a valid command

```
int Active
```

The `Active` property is set to 1 if `command/HicomCommand` contains a valid MML command. This does not imply that it is complete, but HiPath 4000 Expert Access-MML Editor has "something to work with".

The functions `getParam`, `getParamDesc`, `getParamName`, `setParam`, `getValue`, `getValueDesc`, `check`, `checkParam`, `getParamProps`, `getParamValidation`, `existsParam`, `scanMask`, `findMask`, `fieldByName` etc. and the read accesses to the properties `command`, `paramCount`, `HicomCommand` result in a runtime error if called when `Active` equals 0.

Define the Hicom variant

```
string HicomVariant
```

In the string `HicomVariant` the variant of the Hicom is defined.

Get a list of installed Hicom variants

```
string [] GetHicomVariants()
```

This function returns a list of Hicom variants as installed with HiPath 4000 Expert Access.

Hicom Language

```
int HicomLanguage
```

The `HicomLanguage` specifies the current MML language of the PBX. 1 = german, 2 = english. Please refer to the description of `HicomCommand` for further details.

Application language

```
int ApplicationLanguage
```

The `ApplicationLanguage` specifies the current MML language as used in the CBL application. It defines the language for parsing the MML commands as stored in `command`. 1 = german, 2 = english. Please refer to the description of `HicomCommand` for further details.

Hicom command

```
string HicomCommand
```

If `HicomLanguage` and `ApplicationLanguage` differ, the attribute `HicomCommand` can be used for an automated translation of the current MML command. If `HicomCommand` is assigned a MML command in `HicomLanguage`, it is automatically translated to `ApplicationLanguage` and stored into `command` (and vice versa). Please note that `HicomCommand` always uses `HicomLanguage` and `command` always uses `ApplicationLanguage` for MML parsing.

These attributes can be used to support Hicom language independent applications.

Example:

```
comedit.HicomLanguage=1;          // Hicom's language is german
comedit.ApplicationLanguage=2; // CBL-App uses english Parameters
string result=comwin.Send("REGEN-SCSU;");

string [] commands=sys.prepare_regen_results(result);
comedit.HicomCommand=commands[0]; // give first german Hicom result
                                // to comedit.
string first=comedit.command; // first now contains the command in
                                // english.
```

Define keyword or position oriented command output

```
int KeywordOriented
```

If `KeywordOriented` is set to **0** the MML command which is read from property `command` will be position oriented. if `KeywordOriented` is **1** it will be position oriented. If the value is **2**, the command is kept as it was written.

Destination processor

```
string DestinationProcessor
```

`DestinationProcessor` returns the name of the Hicom board where the AMO is designed to be executed. The value can be "SWU", "ADP", "VM", "TD" or "SS7". Please note that the destination processor can be defined at runtime, too.

Number of parameters

```
int ParamCount
```

The `ParamCount` property returns the number of the defined parameters of the AMO-command. The AMO-command is defined in the property `command`, which is described above. `ParamCount` is a read-only property.

Get the value of the parameter (by name)

```
string getParam(string param, int language=0)
```

The `getParam()` function returns the value of a parameter of an AMO. The string `param` is the name of the parameter. The AMO-command must be defined in the function `command`, which is described below. If the parameter does not exist `Error` is set to the appropriate error number.

If `language` equals 0 the value of the parameter is returned in the current `ApplicationLanguage`. If `language` equals `MML_german` (1) or `MML_english` (2) the value is translated to the desired language.

Description of a parameter

```
string getParamDesc(string param, int language=0)
```

The `getParamDesc()` function returns a short help text of the parameter `param`.

If `language` equals 0 the description of the parameter is returned in the current `ApplicationLanguage`. If `language` equals `MML_german` (1) or `MML_english` (2) the description is translated to the desired language.

Set the value of the parameter (by name)

```
int setParam(string param, string value, int language=0)
```

The `setParam()` function sets the value of a parameter of an AMO. The string `param` is the name of the parameter and `value` is the value of the parameter. The AMO-command must be defined in the function `command`, which is described below. If the function fails (parameter not found in current parameter tree or MML syntax error) `Error` will be set to the appropriate error number. The result value is equal to `Error` (0 if ok). The function `CheckParam` is called internally.

If `language` equals 0 the value for the parameter is expected in the current `ApplicationLanguage`. If `language` equals `MML_german` (1) or `MML_english` (2) the value is expected in this language.

Get an allowed value of a parameter

```
string getValue(string param, int index, int language=0)
```

The `getValue()` function returns an allowed value of the parameter `param`. The number of the values is `index`, which is 0-based. If there is no value with the number `index` the returned string is empty.

If `language` equals 0 the allowed value of the parameter is returned in the current `ApplicationLanguage`. If `language` equals `MML_german` (1) or `MML_english` (2) the allowed value is translated to the desired language.

Description of an allowed value

```
string getValueDesc(string param, int index, int language=0)
```

The `getValueDesc()` function returns a short help text about an allowed value of the parameter `param`. The value is defined with the number `index`, which is 0-based.

If `language` equals 0 the value description of the parameter is returned in the current `ApplicationLanguage`. If `language` equals `MML_german` (1) or `MML_english` (2) the value description is translated to the desired language.

Get the name of a parameter at a distinct position

```
string getParamName(int index)
```

This function retrieves the name of the parameter of the current command at position `index`, where `index` is 0-based.

Get the index of a parameter by name

```
int getParamIndex(string name)
```

This function returns the index of parameter `name` within the current command. If the parameter is not found `Error` is set.

Get the names of all parameters in the current command

```
string [] getParamNames()
```

This function retrieves the names of all parameters of the current command.

Get parameter properties

```
int getParamProps(string param)
```

This function currently returns 0 for optional parameters 1 for required parameters (Bit0). The other bits of the return value are reserved for further use.

Get the value of the parameter (by index)

```
string getParamByIndex(int idx)
```

The `getParam()` function returns the value of a parameter of an AMO. The integer `idx` is the number of the parameter. The AMO-command must be defined in the function `command`, which is described above.

Set the value of the parameter (by index)

```
string setParamByIndex(int idx, string value)
```

The `setParam()` function sets the value of a parameter of an AMO. The integer `idx` is the number of the parameter and `value` is the value of the parameter. The AMO-command must be defined in the function `command`, which is described above. If the function fails an error message will be returned.

Get the value of the parameter (by index)


```
string getParamByIndex(int idx)
```

The `getParam()` function returns the value of a parameter of an AMO. The integer `idx` is the number of the parameter. The AMO-command must be defined in the function `command`, which is described above.

Set the value of the parameter (by index)

```
string setParamByIndex(int idx, string value)
```

The `setParam()` function sets the value of a parameter of an AMO. The integer `idx` is the number of the parameter and `value` is the value of the parameter. The AMO-command must be defined in the function `command`, which is described above. If the functions fails an error message will be returned.

Get complete validation of a parameter

```
string getParamValidation(string param, int language=0)
```

This function returns a string that can be passed to a object of class `amoParam` (defined in `pdb.cbl`). This object then contains all information about the Parameter (name, description, type, values, linkage). This object is designed to be used for the form control `ctAmoParameter`.

Error number

```
int Error
```

If the function `Check()` - which is described below - detects an error, the variable `Error` will be the error number.

CBL - Programmer's Reference

Interfaces to the HiPath 4000 Expert AccessFrame components

(mode|=2: application handles wrong password, open returns immediately; mode|=4:

Error	Meaning
1	Reserved
2	Invalid character
3	End of input field
4	Start of input field
5	Input field full
6	Maximum number if individual values already marked
7	Too few individual values marked
8	Mandatory parameters need a value
9	Individual values may not contain blanks
10	Individual value contains odd number of quotation marks
11	First and last characters must be quotation marks
12	Individual value contains invalid length
13	Too few individual values
14	Too many individual values
15	Too many group parameters entered
16	Too few group parameters entered
17	Last parameter
18	First parameter
19	Invalid function key
20	Invalid individual value
21	Internal error
22	Output mask not found
23	Some values could not be assigned
24	Invalid value for branching parameters
25	Value too long
26	Too many parameter values
27	'Command' not set (verb/noun combination unknown)
28	No mask information in the PDB file
29	Output mask could not be analyzed
30	Input file could not be opened
31	Output file could not be opened

Error	Meaning
32	Invalid file position
33	Noun syntax incorrect
34	No MML command
35	PDB file not found
36	Verb not found in PDB file
37	Field not found in mask
38	End of AmoResult reached
40	Parameter not found
41	Index outside valid range
42	Parameter has no defined value list
43	Internal error during parameter check
44	Output mask not found
45	Hicom variant not found (check installation)
46	Syntactical error in GENDB format (faulty headers)
47	Value range links not permitted with 'And-And'
48	Command not terminated with semicolon
49	Link not permitted with 'And-And'
50	Parameter assigned several times
51	Combination of positions and keyword-oriented parameters not permitted
100	End of input file
500	Command buffer is empty
501	Command buffer emptied
502	Entry cannot be interpreted as MML command
503	An error occurred when sending back to HiPath 4000 Expert Access-Connect deactivate autologon.)

With every call to a function or assignment to a property of the HiPath 4000 Expert Access-MML-Editor interface the properties `Error`, `ErrorParam`, `ErrorMsg` and `ErrorParamIdx` are set to the appropriate value.

Error message

```
string ErrorMsg
```

If the check function - which is described below - detects an error, the error message is copied into the string `ErrorMsg`.

Possible error messages: see list above.

Error location

```
string ErrorParam
int      ErrorParamIdx
string ErrorValue
```

The `ErrorParam` property holds the name of the parameter, which is causing an error, `ErrorParamIdx` it's position (as defined in the position oriented view). `ErrorValue` contains the value that caused the error, if available.

Check all the parameters in current command

```
int Check()
```

The `Check()` function checks, if the value range, the syntax and declarations of all the parameters are correct. If an error is detected, the function returns the error number, otherwise it returns 0. The properties `Error`, `ErrorParam` and `ErrorMsg` are set to appropriate values.

Check a distinct parameter

```
int CheckParam(string param)
```

The `CheckParam()` function checks, if the value range, the syntax and declarations of the parameter `param` are correct. If an error is detected, the function returns the error number, otherwise it returns 0. The Function also sets `Error`, `ErrorMsg` and `ErrorParam` to the accurate value.

Check the existence of a parameter

```
int ExistsParam(string param)
```

This function returns 1 if the parameter `param` is defined in the current command, else it returns 0.

Show dialog window for a validated parameter

```
int ShowParamDlg(string param)
```

The `ShowParamDlg()` function creates a dialog window with the parameter `param`. `ComEdit`'s property `command` must have been set to an appropriate command beforehand. If you now make changes to the parameter the return value is 1 and the resulting command is returned in property `command`, otherwise the return value is 0 and property `command` remains unchanged.

Show dialog window for a command

```
string ShowCommandDlg(string _command)
```

ShowCommandDlg opens a modal dialog window for the given parameter `_command`. If the dialog window is closed with "Ok" ShowCommandDlg returns the user entered command and the property `command` is set. If the Dialog is closed with "Cancel" ShowCommandDlg returns `_command` and the property `command` is unchanged.

Transfer a command from another OLE server to HiPath 4000 Expert Access-MML-Editor

This method is designed for use by other OLE servers only, not for use within CBL applications!

```
void ShowCommand(int ConnectionId,  
                 String ConnectionName,  
                 String MMLCommand,  
                 String MMLVariant,  
                 int MMLLanguage,  
                 String ReplyObject,  
                 String ReplyMethod);
```

ShowCommand passes the command `MMLCommand` to the current HiPath 4000 Expert Access-MML-Editor window. If there is no HiPath 4000 Expert Access-MML Editor window existing, a new one will be opened. The HiPath 4000 Expert Access-MML-Editor window will be brought to front. `MMLVariant` tells HiPath 4000 Expert Access-MML-Editor the Hicom variant, `MMLLanguage` (1=german, 2=english) the Hicom command language of the command to be displayed.

If `ConnectionId` is (used defined) non-negative and `ConnectionName` not empty a menu item "Send back to " plus `ConnectionName` will be available from the HiPath 4000 Expert Access-MML-Editor menu "Command". If this menu item (or Enter or F2 or Shift+F2) is executed HiPath 4000 Expert Access-MML-Editor will create a temporary OLE-connection to the OLE-object named `ReplyObject` (e. g. "ComWin.Control") and executes an OLE method named `ReplyMethod` (e. g. "SetCmd") passing `ConnectionId` as first, and the current MML command as second parameter. Then the OLE-connection will be released.

Create a copy of a Comedit object

```
void CopyFrom(Comedit @src)
```

An exact copy of a Comedit object can be created via `CopyFrom`. All relevant information, such as `HicomLanguage`, `ApplicationLanguage`, `command`, `HicomVariant`, etc. is included by the source object `src`.

Parsing AMO display masks

The following procedures/properties can be helpful for parsing AMO results that have been generated by the action ABFRAGEN/DISPLAY of an AMO. The usage of this functionality requires extensive knowledge of the internals of AMO programming and should hence be avoided if possible. **Try to use REGEN-results** instead !

First the "raw" AMO output is assigned to `comedit.AmoResult`.

Please note that `comedit` must know which AMO has sent the output. The following example shows how this can be achieved:

```
comedit.command="DISPLAY-DATE;" ;
comedit.AmoResult=comwin.Send(comedit.command);
```

Now an internal cursor needs to be positioned to the start position of an AMO output mask. This is done with `SkipLinesUntil(string pattern)`:

```
comedit.SkipLinesUntil("+---");
```

Now the internal cursor points to the beginning of the line containing this pattern.

The position of the internal cursor can be read and set via `AmoResultPos`. The value always relates to the character position within the current `AmoResult`.

In the next step we can try to match the current output with an AMO output mask (as defined in the AMOs PD file) by calling `ScanMask(string maskname, int wrparaound=1, int skip=0)`. `ScanMask` returns the name of the output mask which could match the output or "" if there was no matching output mask found. The user can define a mask which should be used for the first try. If this mask did not match, `ScanMask` tries to match all output masks of the AMO. If the optional parameter `wrparaound` is set to 0, `ScanMask` does not try all masks of the AMO, but only from the mask `maskname` to the end of the PD file. If `skip=1` was entered, `ScanMask` does not start at `maskname` but at the next

```
string mask=comedit.ScanMask("abfrdate");
```

The properties `Error` and `ErrorMessage` are set by `ScanMask`, `SkipLinesUntil` and `FieldByName` in case of an error.

If `comedit.ScanMask` returned a valid mask name, `FieldByName(string fieldname)` or `FieldByIndex(int index)` can be used to read the values of the variable fields in the current output mask. The `int` property `FieldCount` is set to the number of variable fields in the current output mask.

Example:

```
string year=comedit.FieldByName("year");
string monat=comedit.FieldByName("month");
int fieldCnt=comedit.FieldCount;
```

The example trace.cbl gives another example where ScanMask is used to parse an AMO output. In this example it was necessary to use ScanMask because the values needed by the application were not available with REGEN commands.

Using the SkipLines(int cnt) function, a defined number (cnt) of lines can be skipped in the AmoResult.

Using the GetNextMask(string maskname) function, the name of the mask after the mask-name mask can be checked in the PD file.

As an alternative the function **FindMask** can be used to find a specific mask in AmoResult. FindMask tries to find the desired mask specified by the parameter maskname starting from the current position within AmoResult. All lines of AmoResult that do not match the desired mask are skipped. In case of an error (e.g. end of AMO result reached) the error variable Error is set and the error code will be returned. If mask is found the return value is 0. The current field values can be read by means of FieldByName.

Example:

```
// Tell comedit to parse AMO BCSU
comedit.command="DISPLAY-BCSU:TAB,1,1;";

// Send the command to Hicom
string result=comwin.Send(comedit.HicomCommand);

// Give the result to Comedit
comedit.AmoResult=result;

// endless loop
for (;;)
{
    // search next occurrence of mask 'tabvar' in AmoResult
    int result=comedit.FindMask("tabvar");

    if (result==0) // if found
    {
        // extract a field from the mask
        system.ConsolePrint("soll="+comedit.FieldByName("soll"));
    }
    else // some error occurred, e.g. end of amo result
    {
        // leave the loop
        break;
    }
}
```

The third alternative available is the function **ParseAmoResult()**. This function analyzes the complete current **AmoResult** and tries to cover as many **AmoResult** lines as possible with AMO masks. The result of analysis is returned as a return value in a suitable array.

As the algorithm is, understandably, not trivial and recursive, the **AmoResult** to be parsed should not be too large.

The elements of the result array are defined as follows:

```
class amoParseMaskRecord
{
    string maskname;
    string skipped;
    string [] fields;
}
```

The following example should clarify the function:

```
include "OpenConnDlg.cbl"
amoParseMaskRecord [] parseResult;
void PrintParseResult()
{
    int i;
    int f;
    comwinFrame.ConsolePrint("-----");
    for i=0 to parseResult.count-1 do
    {
        with (parseResult[i])
        {
            if (skipped!="")
                comwinFrame.ConsolePrintStyle("Skipped "+skipped,cpsWarning);
            if (maskname!="")
            {
                comwinFrame.ConsolePrint("Found mask "+maskname);
                for f=0 to fields.count-1 do
                {
                    comwinFrame.ConsolePrint("  field "+fields[f].index
                                                +" = '"+fields[f]+'');
                }
            }
        }
    }
    comwinFrame.ConsolePrint("-----");
}
```



```
void main()
{
    // tell comedit that we are using english AMOs within our Application
    comedit.ApplicationLanguage=MML_english;
    // tell comedit that we want to work with AMO VEGAS
    comedit.command="DISP-VEGAS;";
    // execute the AMO (HicomCommand returns the command
    // in the AMO language of the Hicom
    string result=comwin.Send(comedit.HicomCommand);
    // give comedit the AMO output
    comedit.AmoResult=result;

    parseResult=comedit.ParseAmoResult();

    PrintParseResult();
}
OpenConnDlg conn;
if (conn.show())
{
    main();
}
```

Additional functions

The function `ShowHelp` causes a html file being displayed in the configured web browser (e.g. Netscape). The file name passed to `ShowHelp` is relative to the Service Handbook Base Url as defined for a specific Hicom Variant. This function can be used to display a specific file of the Service handbook (e.g. `ad1sa.htm`).

File-related check function

The function `CheckFileUntilError` can be used to do a high performance check and transformation of MML batch files.

Parameter `inFile` specifies the name of the input file, parameter `outFile` specifies the name of the output file. It can be set to "" if the output of an check should not be stored.

`StartPos` defines the start position where `CheckFileUntilError` should start (absolute byte position from beginning of the file).

`MaxLines` specifies the maximum of number of lines to be processed until the function should end.

`Action` specifies how the input file should be processed. The value can be:

0 = quick-check; keep MML format.

1 = quick-check, transformation to position oriented MML format

2 = quick-check, transformation to keyword oriented MML format

8 = extensive check, keep MML format.

9 = extensive check, transformation to position oriented MML format

10 = extensive check, transformation to keyword oriented MML format

`HicomLanguage` **and** `ApplicationLanguage` must be set to the language of the MML batch file.

The following example shows how the function can be used to do syntax check of a MML batch file in steps of 1000 commands:

```
include "sysfunc.cbl"
include "comwin.cbl"
int result;
comedit.FilePos=0;
int lineno=0;
string line;
do
{
```

```
result=comedit.CheckFileUntilError("Test.MML",
                                   "",
                                   comedit.FilePos, 1000, 8);

lineno=lineno+comedit.ErrorLine;
line=lineno;
if (result)
{
    system.ConsolePrintStyle("Line "+line+": Error="+comedit.ErrorMsg
                             +"(parameter "+comedit.ErrorParam+)", cpsError);
}
else
{
    system.ConsolePrint(line + " lines processed ...");
}
} while (result!=33);
```

The property `FilePos` always points to the current read position within the input file.

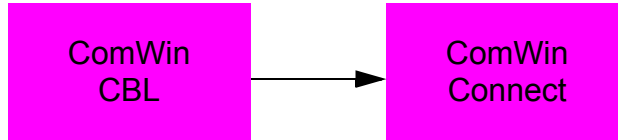
The property `ErrorLine` points to the line number of the current processed command within the input file.

The property `ErrorValue` shows a parameter value which caused the error, if available.

The result value of `CheckFileUntilError` is the current `Error` number.

4.3.2 ComwinControl (ole_class)

Interface to HiPath 4000 Expert Access - Connect.



```

ole_class ComwinControl : "ComWin.Control"
{
  int    Open (string host,
               int mode=0,
               string FamosPassword="",
               int conn=0) ⇨
  string Logon(int deactAFR=0,
               int discVariant=0,
               string userid="",
               string password="",
               int deactSIT=0)⇨
  int OpenDirect ( ... )⇨
  int HTSOpen    ( ... )⇨
  void Attach(string ConnId, int keepopen=0)⇨
  void Show() ⇨
  void Hide() ⇨
  void BringToFront() ⇨
  int WindowState ⇨
  string [] Hosts() ⇨
  string [] Connections() ⇨
  string [] GetParameters(string host) ⇨
  string Logoff() ⇨
  string Send (string cmd) ⇨
  string SendEx(string cmd,int @retcode, string [] conditions,
                int count=1, int timeout=60000)⇨
  string Break() ⇨
  string Print (string cmd)⇨
  string Message (string cmd) ⇨
  string Alert (string cmd) ⇨
  string NewLine () ⇨
  void Close () ⇨
  void CloseAll () ⇨
  int Status ⇨ : "ConnStatus"
  int _Busy ⇨ : "ConnectionBusy"
  string LastMessage ⇨
  int MacroRunning ⇨

```

```
Comedit mml
string HicomLanguage ➡
string HicomVariant ➡
string HostName ➡
string HostDispName ➡
string HostDesc ➡

int    OpenDialog(int mode=0) !!!
void    StartBrowser(string url) !!!
}
```



Please note that the interface to CBL applications is slightly different to applications that connect immediate using the OLE interface. Please see module comwin.cbl to investigate the mapping between CBL- and OLE interface.

Predefined object for this class

ComwinControl comwin;

Open a connection to a host (via LAN connection directory)

```
int Open(string host, int mode=0, string FamosPassword="", int conn=0)
```

The `Open` function opens a connection to a host. The string `host` is the name of a connection profile defined in HiPath 4000 Expert Access-Connect.

The optional parameter `mode` can be used to determine the behavior on opening the connection:

`mode=0`: application handles wrong password, open returns immediately; `mode=4`: deactivate autologon.) (`mode=2`: application handles wrong password, open returns immediately; `mode=4`: deactivate autologon.)

mode	Meaning
1	The connection window is hidden. It can be made visible via the HiPath 4000 Expert Access-Connect main window.
2	No password is requested.
4	The Autologon property from the connection profile, if present, is ignored.
8	No dialog windows are displayed on connection errors.
16	The connection to HiPath 4000 Expert Access-Connect is retained on releasing the OLE reference, i.e. the dialog window remains open.

immediately; `mode=4`: deactivate autologon.)

The optional parameter `FamosPassword` is reserved for further use.

The connection type can be set via the optional parameter `conn`. If `conn` equals 0 the default connection as defined in the connection profile will be opened. If `conn` equals 1 or 2, a LAN connection or a serial connection will be opened.

(mode|=2: application handles wrong password, open returns immediately; mode|=4:

conn	Meaning
0	The connection type is set on the basis of the "Standard" connection in the definition of the connection profile.
1	Connection to FAMOS is set up via LAN/PPP.
2	Connection to RMX is set up via "serial".
8	Connection to Telnet interface.

deactivate autologon.)

If the functions fails it will return 0, otherwise 1.

Open a connection to a host without entry in the connection directory

```
int OpenDirect (  int mode,
                  string HicomId,
                  string Name,
                  string Systemvariant,
                  int Language,
                  int Interface=0,
                  string Line="famos",
                  string Phonenummer="",
                  string Port="23",
                  string userid="",
                  string password="",
                  int AutoLogon=0,
                  string SrcName="FAMOS1",
                  string DstName="UFAMOS",
                  string PdbHost="",
                  int ServerId=0,
                  string SessionToken="",
                  string TemplateProfile="")
```

Using `OpenDirekt` a connection can be opened without a connection profile having first been created in the connection directory. All connection-relevant data must be transferred as call arguments.

mode	See Open ➡
HicomId	Random name for the connection
Name	Random name for the connection
Systemvariant	Hicom system variant (optional)
Language	Hicom AMO language (optional)

Interface	1	Connection via LAN
	0x40	Connection to HiPath 4000 via LAN
Line	famos	Connection to Famos
	telnet	Connection to Telnet
PhoneNumber	Phone number or IP address, depending on the line type	
Port	102	For connections to Famos
	23	For connections via Telnet
UserId	User ID for registration	
Password	Password for registration	
AutoLogon	Perform automatic logon	
SrcName	TSAP-SRC for connections to FAMOS (see AMO CPTP)	
DstName	TSAP-DST for connections to FAMOS (see AMO CPTP)	
PdbHost	Host address from which PDB files can be downloaded	
ServerId	ID of the ComWinAccessServer object in the case of connections to HiPath 4000 (interface=0x40)	
SessionToken	SessionToken for registration at HiPath 4000 (interface=0x40)	
TemplateProfile	Optional connection profile from the connection directory from which all non-specified arguments and other connection information should be transferred.	

Open a connection to a host via a HTS 4.0/4.1, HTS 6 or HiPath Manager

```
int HTSOpen    (string ServerAddress,
               string ServerName,
               string Region,
               string DatabaseUser,
               string DatabasePassword,
               string Token,
               string ServerPath,
               string HicomId,
               string Application,
               string DSNHTSDB,
               string DSNCorrDB,
               int    synchrone,
               string ParameterString,
               string HicomPassword,
               string Version,
```

CBL - Programmer's Reference

Interfaces to the HiPath 4000 Expert AccessFrame components

```
string SubDomain,  
string PdbHost)
```

Using this function an indirect connection can be set up to an application via a HTS server or HiPath Manager.

Call parameter meaning:

ServerAddress	IP address of the HTS server/HiPath Manager	
ServerName	Name of the HTS server/HiPath Manager (for display only)	
Region	HTS server/HiPath Manager region	
DatabaseUser	User ID for registration at database (HTS 4.0/4.1)	
DatabasePassword	Password for registration at database (HTS 4.0/4.1)	
Token	For HTS 4.0/4.1: UDSC token, for HTS 6.0 SessionToken	
ServerPath	-	
HicomId	ID of Hicom to which the connection should be set up (database field chdmain.mnemonic)	
Application	-	
DSNHTSdb	Name of the ODBC DSN for the HTS DB (HTS 4.0/4.1)	
DSNCorrDb	Name of the ODBC DSN for the Corr DB (HTS 4.0/4.1)	
synchronone	If synchronone = 1, the function only returns after complete connection setup. Otherwise, connection setup is only started.	
ParameterString		
HicomPassword		
Version	V40	HTS 4.0
	V41	HTS 4.1
	V60	HTS 6.0/HiPath Manager
SubDomain	-	
PdbHost	Address from which PDB files can be downloaded. The <code>Server-Address</code> can be entered here in the case of HTS 6.0/HiPath Manager.	

"Attach" an application to an existing connection

```
void Attach(string ConnId, int keepopen=0)
```

Using this function a CBL application can be "attached" to an already open dialog connection and then send commands to this connection with `Send`.

A valid connection ID must be transferred as the `ConnId`. This is transferred as command line argument `Args["Id"]` from a connection window by HiPath 4000 Expert Access-Connect when starting up a CBL application via "Start CBL" or via "CBL Bookmark" (see "*Command line parameters: Args*" on page 71).

Alternatively, the connection currently open can be checked via `Connections()` ➡.

If `keepopen!=0` is set, the connection remains open despite the fact that `Close` was called.

Close the connection to a host

```
void Close()
```

The `Close()` function closes the current connection to a host.

Close all connections

```
void CloseAll()
```

The `CloseAll()` function closes all the open connections.

Status of the connection

```
int Status
```

```
int _Busy
```

```
int MacroRunning
```

The `Status` property returns the current connection state.

Bit	connection type	description
Bit0 (LSB) (value 1)	LAN	partner exists
	Serial	COM port exists physically and is not locked by another application
Bit1 "opened" (value 2)	LAN	partner exists
	Serial	"there is something outside", probably a hicom (not sure)
Bit2 "connected" (value 4)	LAN	partner exists
	Serial	if there were digits stored in the connection profile, the modem has connected
Bit3 "logged on" (value 8)	LAN	logon successful
	Serial	

CBL - Programmer's Reference

Interfaces to the HiPath 4000 Expert AccessFrame components

Bit	connection type	description
Bit4 "free" (value 16)		the connection is not being used by another CBL application.

Please note that only bit 3 is unset if the connection state gets worse (e.g. due to a connection loss). Only Hangup or Logoff can be recognized by HiPath 4000 Expert Access-Connect.

The property `_Busy` is 1 if the HiPath 4000 Expert Access-Connect is not yet ready to edit a new AMO command with `Send`, e.g. because a command entered by the user is still being edited.

The property `MacroRunning` is 1 if a macro is being executed on the current connection. Remark: A macro can also be started via `comwin.Send("MACRO datei.cmd")`.

Last error message in a connection

```
string LastMessage
```

If error messages were output during connection setup to the system, the last error message is saved in the property `LastMessage`.

Log on to a host

```
string Logon( int deactAFR=0,  
              int discoverVariant=0,  
              string userid="",  
              string password="" )
```

The function `Logon` enables you to send the logon command to the host to which you are connected.

If the parameter `deactAFR` is non-zero, AMO AFR is regenerated, and the `ACTIVATE` command is stored. AFR is then deactivated.

If parameter `discoverVariant` is non-zero the `Hicom-Variant` and `Hicom-Language` is determined and stored into the property `mml.HicomVariant` and `mml.HicomLanguage` and into the properties `HicomVariant` and `HicomLanguage`.

if parameter `userid` (for "New password concept" since Hicom EV2.0 WP3) and `password` is passed to `Logon`, HiPath 4000 Expert Access-Connect uses these parameters for the logon instead of using the password/userid from the connection profile or prompting with a `userid/password` Dialog

If the functions fails an error message will be returned:

"!WR_PWD" = Wrong Hicom password,
"!WR_VAR" = Hicom release could not be retrieved or is not known to HiPath 4000 Expert Access-MML-Editor,
"!NO_HD" = Hicom hard disk is switched off and cannot be activated.

Log off from the host

```
string Logoff()
```

The `Logoff()` function enables you to logoff from the server, to which you are connected.

If AFR was deactivated in the corresponding `Logon` command the stored ACTIVATE-AFR commands will be sent in advance to the Logoff-sequence.

Send a command to the host

```
string Send(string cmd)
```

The `Send()` function sends a command to the Hicom. The result of the `Send` function is the "piece of protocol" which is the result of the command execution until the host expects user input again (e.g. waiting for next command or prompting a parameter value).

Send a command asynchronously to the host (without waiting for termination)

```
string SendEx(string cmd, int @retcode, string [] conditions,  
              int count=1, int timeout=60000)
```

The `SendEx()` function sends a command to the Hicom.

The result of the `Send` function is the "piece of protocol"; this is the result of the command execution until the point at which `count` conditions from the `conditions` connection array are fulfilled or until the `timeout` (milliseconds) occurred.

The number is 0 at `retcode` in the event of the timeout, otherwise the number of applicable conditions is transferred.

The conditions are to be formulated as "regular expressions" (see Unix command `egrep`).

Example:

```
comwin.Open("80x73");  
comwin.Logon();  
comwin.Send("CHA-FUNCT:SLANG=ENG;");  
int rc;  
string result=comwin.SendEx(  
    "START-REGEN:USERCODE=\"UPT\",SYSNO=\"12345678901234567\",CONT=N;",  
    rc, // returncode = number of fulfilled conditions,  
        // 0 if timeout occurred  
    {"F[0-9][0-9]","H17"}, // list of expected strings
```

CBL - Programmer's Reference

Interfaces to the HiPath 4000 Expert AccessFrame components

```
1,          // number of expected strings to be returned by the AMO.
           // If condition fulfilled SendEx returns immediately
10000 ); // Timeout in milliseconds
comwinFrame.ConsolePrint("rc="+_string(rc)+", result="+result);
if (sys.strfind(result, "H17") == -1)
{
    comwinFrame.ConsolePrint("Error running Full Regen AMO command!");
}
comwin.Close();
```

Print a string in the protocol window

```
string Print(string cmd)
```

The `Print()` function prints the string `cmd` in the protocol window.

Contents of the status line

```
string Message(string cmd)
```

The `Message()` function defines a text (`cmd`) for the status line of a window.

New window for a message

```
string Alert(string cmd)
```

The `Alert()` function opens a message dialog with an (important) message which is defined in the string `cmd`.

Append newline in protocol window

```
string NewLine()
```

With the `NewLine()` function a new-line is appended to the log file.

Set interrupt signal to AMO/MMI

```
string Break()
```

`Break` sends an interrupt signal to Hicom to stop AMO execution or parameter prompting.

Get Hicom variant / version of current connection

```
string HicomVariant
string HicomLangauge
```

Using `HicomVariant/HicomLanguage` the `HicomVariant` and `HicomLanguage` can be retrieved for the current host. The properties are set if `Logon` was called with `discoverVariant=1` (from an CBL application only). If the logon was done without discovering the Hicom version the value is taken from the last time it was set. The value is stored in the registry database at the hosts data. It is accessible via the connection profile directory of HiPath 4000 Expert Access-Connect too.

Get host name and description of current connection

```
string HostName
string HostDesc
string HostDispName
```

`HostName` retrieves the host name of the current connection. `HostDesc` retrieves the description of the current connection, `HostDispName` retrieves the name of the connection as it appears in the title line (e.g. 8073@HTS1).

Change dialog window display

```
void Show()
void Hide()
void BringToFront()
int WindowState
```

A hidden dialog window can be displayed with `Show`.

The current dialog window can be hidden with `Hide`.

The current window can be displayed on top with `BringToFront`. The Windows 2000 operating system does not support this function, however.

If `WindowState` is active, the current window is minimized, otherwise it is displayed as normal.

Administer connections and connection data

```
string [] Hosts()
string [] Connections()
string [] GetParameters(string host)
```

The `Hosts` function provides a list of all connection profiles in the form

```
name|Description
name|Description
```

The `Connections` function supplies a list of connections currently open in the form

CBL - Programmer's Reference

Interfaces to the HiPath 4000 Expert AccessFrame components

```
name|connectionId|inUseFlag  
name|connectionId|inUseFlag
```

Here, `name` is the connection name, `connectionId` is a value which can be used for the `Attach` function. The `inUseFlag` specifies whether an application is already connected to this connection.

Using the `GetParameters` function, specific information on the connection profile whose name is specified in `host` is read out. The result elements have the following meaning:

index	Meaning
0	Famos password (new password concept since EV2.0)
1	Reserved
2	User ID (new password concept since EV2.0)
3	Reserved
4	Password (old password concept)
5	Reserved
6	Defined connections: sum of 1=LAN/FAMOS, 2=V.24 serial, 8=Telnet
7	Default connection (1=LAN, 2=V24, 8=Telnet)

4.3.3 (mode|=2: application handles wrong password, open returns immediately;
mode|=4: deactivate autologon.) **ComwinFrame (ole_class)**

Interface to HiPath 4000 Expert Access - Frame application.

HiPath 4000 Expert Access-Frame implements the architectural frame over HiPath 4000 Expert Access. It cares about the http server, the console window, management of execution of CBL applications by OLE requests and will be responsible for communication between CBL applications.



```
ole_class ComwinFrame : "ComWin.Frame"
{
    void ConsoleShow()
    void ConsoleResize(int top,int left, int width, int height)
    void ConsolePrint(string txt)
    void ConsolePrintStyle(string txt, int style)
    void ConsoleClear()
    void ConsoleHide()
    // mode=1 increment progress bar
    // mode=0 deactivate progress bar
    void ConsoleProgress(int mode)

    void StartApplication(string filename, string args)
    void ShowHelp(...) ➡
}
```



Please note that the interface to CBL applications is slightly different to applications that connect immediate using the OLE interface. Please see module comwin.cbl to investigate the mapping between CBL- and OLE interface.

Predefined object for this class

```
ComwinFrame comwinFrame;

// constants for style of ConsolePrintStyle
const int cpsNormal =0; // Font Arial
const int cpsFixed  =1; // Font Courier
```

CBL - Programmer's Reference

Interfaces to the HiPath 4000 Expert AccessFrame components

```
const int cpsWarning=2; // Color blue
const int cpsError  =4; // Color red
```

Show a console window

```
void ConsoleShow()
```

ConsoleShow opens the Interpreters Console Window. Its Size, Position and contents can be controlled with the following functions:

Set position and size of the console window

```
void ConsoleResize(int top,int left, int width, int height)
```

ConsoleResize sets the absolute position and size in pixels of the console window.

Print a text to the console window

```
void ConsolePrint(string txt)
void ConsolePrintStyle(string txt, int style)
```

ConsolePrint/ConsolePrintStyle prints a text followed by a newline to the console window. If txt contains linefeeds the txt is split into multiple lines.

style defines font and color as shown in following example:

```
ConsolePrintStyle("This is \n an Error",cpsError);
ConsolePrintStyle("H01: MESSAGE FROM SWITCH",cpsFixed+cpsWarning);
```

Clear the console window

```
void ConsoleClear()
```

Hide the console window

```
void ConsoleHide()
```

Show/hide a progress bar in the control window

```
void ConsoleProgress(int mode)
```

ConsoleProgress(1) shows a progress bar in the status line of the console window. Each call of this function shifts the progress bar some pixels.

ConsoleProgress(0) hides the progress bar.

Start a CBL application

```
void StartApplication(string filename, string args)
```

This function causes a new instance of the CBL Interpreter to be executed. The filename of the CBL program is passed in parameter `filename`. Arguments can be passed with parameter `args`.

The format for `args` is "par1=val1 par2=val2 par3=val3". These values will be available stored into the `Args` array for the new CBL application (see "*Command line parameters: Args*" on page 71).

Since the new application is started in a separate process `StartApplication` returns immediately.

Show help file

```
void ShowHelp(string basedir,  
              string project,  
              string file,  
              string topic="",  
              string window="")
```

This function shows the help file.

Argument	Meaning
<code>basedir</code>	Base path for help files.
<code>project</code>	The .chm file (compiled HTML help project) or "", if .htm files are to be directly displayed.
<code>file</code>	The name of the HTML file included in the project or to be directly displayed.
<code>topic</code>	The name of an internal cross-reference destination within the HTML file () or "", if the HTML file should be displayed from the start.
<code>window</code>	Optional; a type of window defined, where applicable, in the help project.

If Hypertext-Help-ActiveX is installed (under Win2000/XP) in the current environment, the hypertext help module is called with the following URL:

```
file:/<basedir>/<language>/<project>::<file>#<topic>
```

In addition, the window type defined in the help project (where applicable) is used for branching. <language> is the current language ID, see "*CBL - National Language Support*" on page 174.

CBL - Programmer's Reference

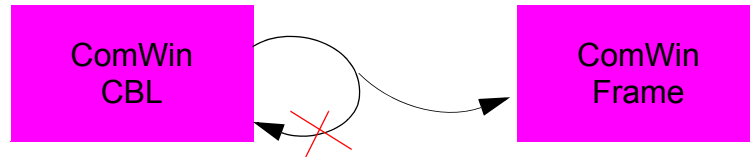
Interfaces to the HiPath 4000 Expert AccessFrame components

If a project file is not specified or HyperHelp is not installed, then the following URL is transferred to the current HTML standard browser:

```
file: /<basedir>/<language>/<file>#<topic>
```

4.3.4 ComwinCBL (ole_class)

Interface to HiPath 4000 Expert Access - CBL Interpreter



An architectural change caused all OLE requests from a CBL Interpreter to itself to be either implemented in Interpreter internal functions or be redirected to HiPath 4000 Expert Access-Frame. HiPath 4000 Expert Access-CBL is just listed here because the chosen implementation is backward-compatible. Consequently, you may find references to HiPath 4000 Expert Access-CBL in existing applications, but they are redirected to ComwinFrame (see CBL include file `comwin.cbl`).

All methods of the predefined object `system` are redirected to `comwinFrame`.

4.3.5 ComWinFT (ole_class)

Interface to HiPath 4000 Expert Access - File Transfer.

Please make sure that you also reference the online help for specific details on the features and configuration of the HiPath 4000 Expert Access file transfer.

This object enables the exchange of files between the CBL application and Hicom or other servers using various protocols.



```
enum { FTPProto_Local=0,
        FTPProto_RmxScsi=1,
        FTPProto_FTP=3,
        FTPProto_FJAM=4,
        FTPProto_HTS=5,
        FTPProto_DispaList=6,
        FTPProto_CORBA=10,
        FTPProto_TFTP };
enum { RmxFileType_ByteFile=0,
        RmxFileType_Fix=1,
        RmxFileType_Var=2,
        RmxFileType_Dir=3,
        RmxFileType_ByteFileOMF=4 };
enum { FT_Success=0,
        FT_UserInterrupt=1,
        FT_Aborted=2,
        FT_LocalFile=3,
        FT_RemoteFile=4,
        FT_ConversionError=5,
        FT_WrongLogon=6,
        FT_TransmissionError=7,
        FT_NotConnected=8,
        FT_Undefined=9,
        FT_NotSupported=10,
        FT_TransferAborted=11,
        FT_HtsError=12 };
```

```
class RmxFileInfo
{
    int FileSize;
```

```
    int FileType;
    int RecordLength;
    string Access;
    string Date;
    string APS;
};

ole_class ComWinFT : "ComWin.FileTransfer"
{
    int SetProtocol(int Proto) ⇨
    string [] GetHosts() ⇨
    int Connect(string Host="", string user="", string pass="",
                string par1="", string par2="",
                string ftuser="", string ftpass="", int batch=0,
                string res1="", string res2="", string res3="",
                string res4="") ⇨
    int Disconnect() ⇨
    string [] GetDir (string Dir) ⇨
    int Get(string Local, string Remote,
            int ConvertSAM=0, int Batch=0, int maxlen=0) ⇨
    int Put(string Local, string Remote, ConvertSAM=0, int Batch=0) ⇨

    RmxFileInfo GetInfo(string Remote) ⇨

    int Delete (string Remote)⇨
    int Rename (string oldname, string newname)⇨

    int    Connected ⇨
    int    Visible ⇨
    int    CloseAfterAutomation ⇨
    int    DialogMode ⇨
    string LocalFile ⇨
    string RemoteFile ⇨
    void    ShowBrowser() ⇨
    int    FileUnsplit(string FileName) ⇨
    int    FileSplit(string FileName) ⇨
}
```

Define transfer protocol

```
int SetProtocol(int Proto)
```

SetProtocol is used to define the desired transfer protocol.

The following transfer protocols may be selected: `FTProto_RmxScsi`, `FTProto_FTP`, `FTProto_FJAM` and `FTProto_DispalList`. Note that the partners for the file transfer depend on which transfer protocol is used. For example, the protocol `RmxScsi` requires the `PCHI` tool to be installed and a SCSI cable to be connected, since the file transfer commands `rmxcopy` and `rmxdirect` are combined here. The `FTP` protocol can be used to establish connections with standard `FTP` servers (e.g. Unix servers). The `FJAM` protocol can be used for LAN connections to the `Hicom` file transfer, which in turn requires the `Hicom` to be properly configured with the `AMOs` `CPTP`, `FTCSM` and `FTRNS`. The `Dispallist` protocol can be used for all connections to `Hicom` systems (serial direct, modem and LAN), since the file transfer is implemented via the `AMOs` `LIST` and `DISPA` here.

Get list of configured communication partners

```
string [] GetHosts()
```

`GetHosts` can be used to determine which communication partners are available for the currently set protocol. This is irrelevant for the `FTP` protocol; for `FJAM` and `Dispallist`, the appropriate partners should be listed in the connection folder of `HiPath 4000 Expert Access-Connect`.

Set up a connection

```
int Connect(string Host="", string par2="", string par3="",
            string par4="", string par5="",
            string par6="", string par7="", int batch=0,
            string par8="", string par9="", string par10="",
            string par11="")
```

`Connect` sets up the initial connection to the communication partner. Depending on which protocol is set, appropriate arguments must be provided to establish the connection.

`Host` specifies the name of the respective partner.

The following overview shows the meaning of the parameters for the various protocols:

`FTProto_DispalList:`

```
Connect( <Host>, <RmxUser>, <RmxPass> );
```

`FTProto_FJAM:`

```
Connect( <Host>, <RmxUser>, <RmxPass>,
        <TsapSrcFt>, <TsapDstFt>, <FtUser>, <FtPass>,
        <batch(ignored)>, <TsapSrcFamos>, <TsapDstFamos>,
        <FamosPort> );
```

`FTProto_FTP:`

```
Connect( <Host-IP>, <FtpUser>, <FtpPass> );
```

```
FTPProto_HTS:
    Connect( <Server-IP>, <HostName>, <Token>, <Region>, <User>,
            <ConnectionId>, <DatabasePass>,
            <batch(ignored)>, <DSNHtsDb>, <DSNCorrDb> , <ServerName>,
            <Version>);

FTPProto_Corba:
    Connect( <Profile(ignored)>, <Host-IP>,
            <Password(ignored)>, <Token> );
```

The return value indicates whether the connection was set up successfully. The return value for a successful setup is 0.

The following return values are defined:

Value	Meaning
1	Aborted by user
2	Operation aborted
3	Error on reading or creating local file
4	Error on reading or creating remote file
5	SAM - ASCII text conversion error
6	Invalid logon
7	Transmission error
8	Not connected
9	Undefined error
10	Operation not supported
11	Transmission interrupted
12	Error at HTS server

Disconnect

```
int Disconnect()
```

`Disconnect` clears the current connection. The return values are the same as those indicated in the table above.

Get contents of remote directory

```
string [] GetDir (string Dir)
```

The function `GetDir` returns the contents of the remote directory `Dir`.

Retrieve a file

```
int Get(string Local, string Remote,  
        int ConvertSAM=0, int Batch=0, int maxlen=0)
```

`Get` can be used to initiate the transfer of a remote file `Remote` to the local file `Local` from a remote system.

`ConvertSAM` specifies whether the remote file is an RMX-SAM file and is to be converted to an ASCII text file.

`Batch!=0` can be used to specify whether the transfer is to occur asynchronously (i.e. in the background) or whether the system should wait for the result of the transfer.

`maxlen` can be additionally used to define if the entire file (`maxlen=0`) is to be transferred or if only a maximum of `maxlen` characters should be received.

Send a file

```
int Put(string Local, string Remote, ConvertSAM=0, int Batch=0)
```

`Put` can be used to transfer the local file `Local` to the remote file `Remote` on a remote host.

`ConvertSAM` specifies whether the remote file is an RMX-SAM file, i.e., whether the local ASCII text file is to be converted during the file transfer to that format.

`Batch!=0` can be used to specify whether the transfer is to occur asynchronously (i.e. in the background) or whether the system should wait for the result of the transfer.

Get file information

```
RmxFileInfo GetInfo(string Remote)
```

`GetInfo` returns information on the file `Remote` on the remote host, provided this function is supported by the selected transfer protocol.

The structure element `FileSize` contains the size of the file in bytes; `FileType` contains the type of file. A distinction is made between the following file types:

```
RmxFileType_ByteFile=0 (binary file),  
RmxFileType_Fix=1 (SAM file with fixed record length),  
RmxFileType_Var=2 (SAM file with variable record length),  
RmxFileType_Dir=3 (directory),  
RmxFileType_ByteFileOMF=4 (binary file in OMF format).
```

`RecordLength` contains the record length of SAM files with a fixed record length.

`Access` contains the access rights, depending on the transfer protocol and type of remote host.

Date contains the file date.

For binary files in OMF format, APS contains the designation of the APS with which the file was produced.

Other attributes and functions

int Delete (string Remote)	Deletes a file on the remote system.
int Rename (string oldname, string newname)	Renames a file on the remote system.
int Connected	Specifies whether or not a connection to the remote system exists.
int Visible	Visible can be used to determine or set if the application is to be visible or not.
int CloseAfterAutomation	CloseAfterAutomation defines whether or not the application is to be terminated after releasing the OLE reference. Default value is 1 (true).
int DialogMode	Can be set to 0 to prevent the dialog window from appearing in the event of a fault. Default value is 1 (true).
string LocalFile	Contains the local file name of the last file transfer.
string RemoteFile	Contains the file name on the remote system of the last file transfer.
void ShowBrowser()	Opens the "remote browser" for the remote system.
int FileSplit (string FileName)	A file on the local file system is split into 1MB blocks. The resulting files are named in the same way as the PCHI tool, i.e. <FileName>.1.part, <FileName>.2.part, etc.
int FileUnsplit (string FileName)	A file that was split with FileSplit is merged back into a single file.

4.3.6 ComWinPPP (ole_class)

This OLE class implements the OLE interface to the HiPath 4000 Expert Access-PPP-Manager. This OLE interface is primarily used by HiPath 4000 Expert Access-Connect to set up connections for which a PPP profile is entered in the LAN data. It can also be used directly by applications.

```
ole_class ComWinPPP : "ComWin.PPPMan"
{
    string Id
    int     Visible
    string Profile
    string PhoneNumber
    string ServerIp
    string Device
    string Script
    string User
    string Password
    string Status
    int     Connected
    string DeviceList
    int     DestType
    int     RemoveProfile(string id, string profile="")

    void     InitProfile(int type)

    int     CreateProfile(string srcid, string dstid, string
srcprofile="", string dstprofile="")
                                : "CreateProfile"

    int     DialogMode
    int     RasConnError
    int     RasConnState
    string RasStatusString
}
```

string Id	Name of the HiPath 4000 Expert Access-Connect connection profile
int Visible	Specifies whether or not the PPP Manager is visible
string Profile	Name of the RAS profile for the current connection profile (the connection profile is prefixed by PPP_)
string PhoneNumber	Phone number of the RAS server

<code>string ServerIp</code>	IP address of the RAS server
<code>string Device</code>	Name of the modem to be used for setting up the connection
<code>string Script</code>	Path of the logon script assigned to the RAS profile
<code>string User</code>	CHAP name for RAS logon
<code>string Password</code>	CHAP secret for RAS logon
<code>string Status</code>	Specifies the current logon status
<code>int Connected</code>	Specifies whether or not a connection is set up to the RAS server. If <code>Connected</code> is set from 0 to 1, the connection setup attempt is started
<code>int DestType</code>	Specifies the type of the connection profile: 0 = connections to UW7 via V.24 for Hicom 300 <= EV3.1 1 = connections via WAML via ISDN 2 = connections to UW7 via V.24 for HiPath 4000 UV1.0 or later 3 = other connection type
<code>int DialogMode</code>	Specifies whether or not message dialogs are to be displayed in the event of faults. Default value 1.
<code>int RasConnError</code>	Error number of the last RAS connection error
<code>int RasConnState</code>	Number of the current RAS connection status
<code>string RasStatusString</code>	Textual value of the current RAS connection status

4.3.7 ComWinAccessServer (ole_class)

The OLE class ComWinAccessServer implements the interface to the ComWinDbAccessClient.exe program. This program represents the interface to the server process ComWinAccess. This process runs in the Unix part of the HiPath 4000 ADP and under HiPath Manager and HTS V6.0. ComWinAccess server and client communicate via CORBA.

See also example *"Database query via ComWinAccess" on page 142.*

```
ole_class ComWinAccessServer : "ComWinDbAccess.Server"
{
    string Open(string host, string port="7777")
    int      Id
}
```

string Open (string host, string port="7777")	This function sets up a connection to the server process. An error message is output in the event of a fault. The <code>host</code> parameter is then given the host name or the IP address of the server.
int Id	In the case of successful connection setup, <code>Id</code> contains a server ID which is needed for the following objects of the types ComWinAccessSession, ComWinAccessFileman, ComWinAccessDatabase and ComWinAccessQuery.

4.3.8 ComWinAccessSession (ole_class)

The OLE class ComWinAccessSession implements the interface to a session to a ComWinAccessServer, see *"ComWinAccessServer (ole_class)" on page 138.*

This object administers all session-related data.

See also example *"Database query via ComWinAccess" on page 142*

```
ole_class ComWinAccessSession : "ComWinDbAccess.Session"
{
    int Id
    int Server
    int Authenticate(string SessionToken)
    int GetFamosUserPass(string @ user, string @ pass)
    string CreateSessionToken()
    string GetInfo(string item)
    int PrintLog(int type, string p1, string p2="", string p3="",
                string p4="", string p5="", string p6="")
}
```

<code>int Id</code>	The <code>Id</code> transferred here should be used for the objects <code>ComWinAccessFileman</code> , and <code>ComWinAccessDatabase</code> for their property <code>Session</code> to link these objects to this session.
<code>int Server</code>	After creating the object, the <code>Id</code> of a <code>ComWinAccessServer</code> object should be sent.
<code>int AuthenticateSessionToken(string SessionToken)</code>	The CBL applications uses this function for authentication at the <code>ComWinAccess</code> server. A valid <code>SessionToken</code> or a string should be sent in the form " <code><user>/*:*/<password></code> ", for example, " <code>engr/*:*/xxx.xxx</code> ". If the <code>SessionToken</code> or the <code>UserId</code> password combination is invalid, then a logon dialog appears (unless <code>DialogMode=0</code>). The system returns 0 in the case of correct functionality, otherwise an error code.
<code>int GetFamosUserPass(string @ user, string @ pass)</code>	In the case of successful registration, this function can be used to call up a valid Famos user ID and the associated Famos password for the current session.
<code>string CreateSessionToken()</code>	Additional <code>SessionTokens</code> can be generated with this function, as every <code>SessionToken</code> is invalid when used first.
<code>string GetInfo(string item)</code>	This function can be used to call up server- or session-related data from the server. The following items are defined: " <code>user_supersess</code> " = user name of the superordinate session " <code>user</code> " = user name for session registration " <code>password</code> " = password for session registration " <code>os_user</code> " = user name on the operating system level " <code>featureList</code> " = the user-assigned features " <code>HomeDir</code> " = home directory of the user on the server " <code>ServerType</code> " = server type (" <code>RM</code> " for RM, " <code>UW</code> " for UnixWare) " <code>ServerVersion</code> " = internal server version
<code>int PrintLog(...)</code>	This function can be used to make entries in the <code>SessionLog</code> . At present, only 0 is implemented as <code>type</code> . <code>p1</code> should be the name of the PBX, with which the action is performed, <code>p2</code> is a short message (30 characters), <code>p3</code> is a "long" message (255 characters).

4.3.9 ComWinAccessFileman (ole_class)

The OLE class `ComWinAccessFileman` implements the interface to the file system of the `ComWinAccess` server, see "*ComWinAccessServer (ole_class)*" on page 138.

This description is incomplete as it should only be used via HiPath 4000 Expert Access-File Transfer in FT_CORBA mode, see *"ComWinFT (ole_class)" on page 130*.

```
ole_class ComWinAccessFileman : "ComWinDbAccess.Fileman"
{
    int Server
    int Session
    int GetFile(string local, string remote)
}
```

int Server	The ID of a ComWinAccessServer object should be transferred here after object creation.
int Session	The ID of a ComWinAccessSession object should be transferred here after object creation.
int GetFile (string local, string remote)	Performs file transfer from the ComWinAccess server to the local file system. The return value corresponds to the HiPath 4000 Expert Access-File Transfer error code.

4.3.10 ComWinAccessDatabase (ole_class)

The OLE class ComWinAccessDatabase implements the interface to a database on the ComWinAccess server, see *"ComWinAccessServer (ole_class)" on page 138*.

See also example *"Database query via ComWinAccess" on page 142*.

```
ole_class ComWinAccessDatabase : "ComWinDbAccess.Database"
{
    int Server
    int Session
    int Id
    int OpenDatabase(string dsn, string user, string pass)
    void CloseDatabase()
    string GetErrorMsg()
}
```

int Server	The Id of a ComWinAccessServer object should be transferred here after object creation.
int Session	The Id of a ComWinAccessSession object should be transferred here after object creation.
int Id	The Id read out here can be transferred in the Database property for one or more ComWinAccessQuery objects in order to link the SQL prompt with the database.

int OpenDatabase (string dsn,...)	The database whose name is specified via dsn is opened with Open-Database. The return value is 1 if the database could be opened, otherwise it is 0 or negative, if no valid session or no database processing authorization is available.
void CloseDatabase()	This closes the database.
string GetErrorMsg()	If an error occurs, an associated error message can be queried with this function.

4.3.11 ComWinAccessQuery (ole_class)

The OLE class ComWinAccessQuery implements the interface to a database query on the ComWinAccess server.

see "*ComWinAccessDatabase (ole_class)*" on page 140

see "*ComWinAccessServer (ole_class)*" on page 138

```
ole_class ComWinAccessQuery : "ComWinDbAccess.Query"
{
    int Server
    int Database
    string ExecSql(string sql)
    string SetParam(int param, int sqltype, string val)
    string GetFirst()
    string GetNext()
    string GetErrorMsg()
}
```

int Server	The ID of a ComWinAccessServer object should be transferred here after object creation.
int Database	The ID of a ComWinAccessDatabase object should be transferred here after object creation. The SQL statement is performed for the database opened there.
String ExecSql (string sql)	Using this, an SQL query or an SQL command can be sent to the currently connected database. Error messages or advisories are returned in the return value.
string SetParam (int param, int sqltype, string val)	Mit dieser Funktion können SQL-Parameter eingefügt werden, die im SQL-Statement noch mit ? vorbelegt waren.

Interfaces to the HiPath 4000 Expert AccessFrame components

Example: Database query via ComWinAccess




```
string queryresult=query.ExecSql("select * from chdmain;");

class cwdbf
{
    int type;
    string data;
}
cwdbf [] record;
int i;

for (queryresult=query.GetFirst(); queryresult!="";
    queryresult=query.GetNext() )
{
    record=queryresult;
    comwinFrame.ConsolePrint("-----");
    for i=0 to record.count-1 do
    {
        comwinFrame.ConsolePrint(record[i].index
                                + "("+_string(record[i].type)+")" + "=" +
record[i].data);
    }
}
database.CloseDatabase();
```

4.3.12 ComWinFM (ole_class)

The OLE class ComWinFM implements the OLE interface to the AfrSvr.exe program. This is an application which is capable of receiving Hicom/HiPath error messages sent from the PBX via LAN. The PBX should be configured accordingly. This is done via the AMOs CPTP, AFR, SIGNL, and FBTID.

With the help of the ComWinFM class, a CBL application can be written and used to analyze and process error messages. An appropriate sample application can be found at the end of this section.

```
ole_class ComWinFM : "ComWin.FM"
{
    void Init()
    int  Visible

    void OnDataAvailable(string sender, string fbtid, string msg)
                                     : "IFMEvents:DataAvailable"
    void WaitForMessages(int sec)
}
```

void Init()	This method should be called up after creating the object to initialize the object.
int Visible	Controls or specifies whether the application should run in the foreground (visible) or in the background.
void WaitForMessages(int sec)	This function is called up to enable the CBL Interpreter to receive events from an OLE server for a specific period of time (sec). See int WaitUntilReleaseWait (int @ context, int timeout)
OnDataAvailable	This is not a function, but rather an event handling routine which is called up when error messages are received. To run a separate code for event handling, a separate class in which the OnDataAvailable function is declared must be derived from the ComWinFM basic class. See also the following example.

Example: Processing error messages further

The following program receives Hicom error messages for a minute and outputs them via the HiPath 4000 Expert Access Frame console window.

```
class myFM extends ComWinFM
{
    void OnDataAvailable(string sender, string fbtid, string msg)
    {
        comwinFrame.ConsolePrint(msg);
    }
}
```

```
    }  
}
```

```
myFM fm;  
fm.Init();
```

```
fm.WaitForMessages(60);
```

4.4 Designing forms

Before you begin reading this section, please note that there is also a "GuiBuilder" tool supplied with HiPath 4000 Expert Access which - though still in a prototype or Beta version - can be used to graphically edit forms using Drag&Drop techniques. For more detailed information, please start the HiPath 4000 Expert Access-GuiBuilder and refer to the online help.

The include file **form.cbl** contains class definitions which are necessary for designing a GUI for a CBL application. Each `cblForm` object may act as a so called main form (representing the whole window) or as an embedded subform (representing e.g. a panel).

Each window consists of a main form which may contain several controls, a menu bar and a speed bar.

There are two groups of controls: Simple controls (label, edit, combo, memo, listbox, radiogroup, checkbox, button, bitmap) and compound controls (panel, pages, grid).

Compound controls are used to group subforms and to get a visual effect (e.g. tab sheets). Of course, a subform cannot have a menu bar or speed bar of its own. These attributes are ignored for subforms.

To build a form, you create an object of class `cblFormControl` for each of your controls and an object of class `cblForm` for each of your main- and subforms. Then, you assign references to the contained controls objects to the `controls[]` attribute of the appropriate form object and you assign references to the contained subforms to the `panels[]` attribute of the appropriate compound control object.

It is highly recommended to assign these references at runtime by means of a `init()` method in your class and not through static initialization!

Advanced topics

- > chapter 4.4.1, "Form Layout"
- > chapter 4.4.2, "The form class `cblForm`"
- > chapter 4.4.3, "Form controls"
- > chapter 4.4.4, "Definition of menus"
- > chapter 4.4.5, "Form examples"
- > chapter 4.4.6, "Modal dialog example"

4.4.1 Form Layout

Currently there are two ways to define the layout of forms: Pixel based layout and cell layout. Here only cell layout is explained, as pixel based layout is considered obsolete.

In defining a layout, you have to specify size (height and width), position (top, left, col, row) and alignment of each control (including compound controls) and form.

You start with simple controls and group them into compound controls until you have defined the complete form.

4.4.1.1 Size of Controls

For simple controls you usually don't specify a size. This causes the CBL-Interpreter to automatically compute the size needed. If you do specify size and/or position you have to specify it in pixels.

If you don't specify the size of a simple control, the CBL-Interpreter determines the size. For text strings (e.g. label captions) the font size, screen resolution and the actual text is considered automatically. This is especially handy, if you think about internationalization of your text strings! Additionally a standard function `TextWidth(string in)` is provided, which computes the length of its string parameter in pixels. Thus, you can use a call `TextWidth("9999")` to define the width of e.g. an edit field which is wide enough to contain four digits.

4.4.1.2 Position of controls

Each (sub-)form consists of an invisible two-dimensional array of cells. Each contained control is placed into one of those invisible cells.

You simply specify the row and column of each contained control in the row and col attribute. Additionally you can specify either an pixel offset within the cell by using the top and left attribute or an alignment of the control within the cell.

The CBL-Interpreter then automatically determines the maximum width and height of all controls in a row resp. column and computes the necessary height and width of the rows and columns and thereby the size of the complete (sub-)form. The algorithm does not make a difference between visible and invisible controls. Thus, the layout does not change if you make controls visible or invisible at runtime. Another effect is, that you can use invisible controls to reserve space on your form and thereby better control the layout of the form.

4.4.1.3 Connecting controls with program variables

tbd.

object of class `cblForm` is usually a member of the corresponding data object;

4.4.2 The form class **cblForm**

```
class cblForm
{
    cblLayout          layout;
    string             caption;
    cblMenu            mainmenu;
    cblMenuItem        speedbar[];
    string             status;
    int                state;
    cblFormControl     @controls[];

    int                enabled=1;
    int                visible=1;
    int                showModified=1;

    int                isMainForm=0;
    string             icon;
    int                helpContext=0;
    int                @breakFlag;

    void               @onEnter;
    void               @onClose;
    void               @onHelp;

    void               Show()
    int                ShowModal()
    void               Update(int relayout=0)
    void               UpdateControl(cblFormControl @ctrl,
                                   int recursive=0)
    int                CheckComplete()
    int                CheckControl(cblSyFormControl @ctrl)
    void               ResetModified()
    void               ResetModifiedControl(cblSyFormControl @ctrl)
    void               Get()
    void               GetControl(cblFormControl @ctrl,
                                   int recursive=0)
    void               Close()
    void               FocusControl(cblFormControl @ctrl)
    void               SetStatus(string text)
    void               SetWindowState(int ws)
    void               StoreSizePos()
    void               RetrieveSizePos()
}
```

<code>layout</code>	defines position and size of the form.
<code>caption</code>	defines the caption (title) of the form.
<code>mainmenu</code>	assigns a menu to the form.
<code>speedbar</code>	allows the definition of a row of speed-buttons (with bitmaps).
<code>controls</code>	is the list of form controls as defined later.
<code>status</code>	is the content of the status line of the form.
<code>state</code>	is used for the form's state regarding to the CBL GUI Styleguide. It's values can be <code>stQuery</code> , <code>stModify</code> , <code>stNew</code> and <code>stShow</code> .
<code>isMainForm</code>	if <code>isMainForm</code> equals 1 the caption of the form is shown in the task bar as application name.
<code>icon</code>	<code>icon</code> points to an .ico file containing the icon for the CBL application.
<code>enabled</code>	defines if the user can interact with a form. Default is <code>true</code> .
<code>visible</code>	defines if the current form/panel is visible. Default value is <code>true</code> . This property is of interest if a form is used as tabsheet within a page control.
<code>showModified</code>	defines if change bars are to be printed automatically. Default value is <code>true</code> .
<code>helpContext</code>	The value of <code>helpContext</code> is passed to the <code>onHelp</code> handler function if F1 is pressed or "Help" is selected from the context sensitive menu.
<code>breakFlag</code>	<code>breakFlag</code> can be used to assign an <code>int</code> variable to a form. This <code>int</code> variable is set to 1 as soon as the user presses the Ctrl+C key combination or executes a "Break" from the context-sensitive menu of the form. The CBL programmer is responsible for initializing and re-setting the assigned variable.
<code>onEnter</code>	is called as soon as <code>form.Show()</code> is called.
<code>onClose</code>	The referred function is called if the user tries to close the form window by means of the window system menu or by clicking the title bar's close button. If the window should be closed the function <code>Close()</code> has to be called. If <code>onClose</code> is not assigned the form will be closed automatically.
<code>onHelp</code>	is a reference to a function which is called if the menu item "Help" is selected from the context sensitive menu (right mouse key) of the form or F1 is pressed. If <code>onHelp</code> refers to a function which accepts an integer value as first parameter the <code>helpContext</code> value of the currently active control is passed to this function.

<code>Show()</code>	<p>is used to display the form for the first time (or for redrawing if the layout has changed). The call will be like <code>form1.Show()</code> ;</p> <p>To close the window later you have to call the <code>Close()</code> method. This is usually done in response to clicking an appropriate button on the form.</p>
<code>ShowModal()</code>	<p>is used to display the form as a modal dialog. The dialog window is closed only when a button of the form is clicked, where all of the following conditions are met:</p> <ul style="list-style-type: none"> • <code>button.style==0</code> • <code>button.tag</code> greater than 0 • if an <code>onClick</code> event handler is defined for the button and the event handler returns an <code>int</code> result, this result must not be equal 0 <p><code>ShowModal()</code> returns the <code>int</code> result of the event handler, if it exists, otherwise it returns the <code>tag</code> value of the clicked button as result value. You never call the <code>Close()</code> method to close a modal dialog.</p>
<code>Update</code> (<code>int</code> <code>relayout=0</code>)	<p>is used to show the values of the designated value variables to the corresponding form controls. It also updates/interprets the values of the options <code>readonly</code>, <code>visible</code> and <code>enabled</code>. If the optional parameter <code>relayout</code> is set to 1 the form is "relayouted" and repainted completely.</p>
<code>UpdateControl</code> (<code>cblFormControl</code> <code>@ctrl, int recursive=0</code>)	<p>Same as <code>Update</code>, but updates only the one Control that is passed as argument <code>ctrl</code>. Example:</p> <pre>form1.UpdateControl(form1.controls["edit"]);</pre> <p>If parameter <code>recursive</code> is true and <code>ctrl</code> is a compound control, then the nested controls are updated, too.</p>
<code>CheckComplete()</code>	<p>can be called to check if all form controls where the attribute <code>required</code> is true(1) got a value other than "". <code>CheckComplete()</code> does not check controls where the <code>visible</code> attribute is false. It returns 1 if the check was successful.</p> <p>The attribute <code>error</code> is set to 1 for each required control with no value.</p>

<code>CheckControl(cblsyFormControl @ctrl)</code>	<p>checks the current value of the associated <code>value</code> variable against the validation specified by <code>items</code>. Only valid for controls of type <code>ctAmo-Parameter</code>.</p> <p>The return value is:</p> <ul style="list-style-type: none"> 0 = no error 1 = required parameter needs a value 2 = spaces are not allowed in individual values 3 = odd number of quotes 4 = quotation mark missing 5 = illegal individual value 6 = ranges may not be concatenated with <code>&&</code> 7 = too few individual values 8 = too many individual value 9 = illegal character in value
<code>ResetModified()</code>	resets the <code>modified</code> attribute of all controls contained in the form and all its subforms and internally stores the current value of each control as the new original value of the control. This original value is used to determine, when the <code>modified</code> attribute has to be set/reset by the CBL Interpreter.
<code>ResetModified- Control(cblsyFormControl @ctrl)</code>	resets just the <code>modified</code> attribute of the control <code>ctrl</code> .
<code>Get()</code>	reads the values of all controls into the corresponding value variables. The values of all controls contained in a form are read, even if they are invisible, disabled or hidden by other controls, panels etc.
<code>GetControl (cblFormControl @ctrl, int recursive=0)</code>	<p>Same as <code>Get</code>, but reads only the value of the one Control that is passed as argument <code>ctrl</code>. Example:</p> <pre>form1.GetControl(form1.controls["edit"]);</pre> <p>If parameter <code>recursive</code> is true and <code>ctrl</code> is a compound control, then the values of the nested controls are read, too.</p>
<code>Close()</code>	is used to close the (nonmodal) window.
<code>FocusControl (cblFormControl @ctrl)</code>	can be used to give the focus to a specific form control. The control must be contained in the current form.
<code>SetStatus (string text)</code>	may be used to update the status field of a form immediately.

<code>SetWindowState</code> <code>(int ws)</code>	sets the display mode of the current form. <code>ws</code> accepts <code>wsHide</code> <code>wsShow</code> <code>wsActivate</code> (gives the current window the focus) <code>wsMinimize</code> <code>wsMaximize</code>
<code>StoreSizePos()</code>	saves the current window position and size in the Windows registry (under <code>HKEY_CURRENT_USER/Software/Siemens/ComWin 300/Layouts/<name and path of the CBL application>/<name of the form (name)>/</code>). This function should normally be called on closing the form. On calling the <code>Show</code> function, the CBL Interpreter first looks for a matching entry for the current window in the Windows registry and displays the window with the saved position and size if such an entry is found.
<code>RetrieveSizePos</code> <code>()</code>	reads the window position and size that was saved with <code>StoreSizePos</code> from the Windows registry and sets the position and size of the current window accordingly if a matching entry was found.

4.4.3 Form controls

Class `cblFormControl` is the common class for all controls. The actual type of a control is determined by attribute `type`. Depending on the actual type of the control some of the attributes of the `cblFormControl` object may be ignored.

In this section only those attributes are explained, which are common to all types of controls.

```
class cblFormControl
{
    int      type;

    int      row;
    int      col;
    int      rowspan=1;
    int      colspan=1;

    int      left;
    int      top;
    int      width;
    int      height;

    int      align=alNone;
    string    caption;
    string    hint;
    int      enabled=1;
```

```

int      readonly=0;
int      required=0;
int      visible=1;
int      outofdate=0;
int      modified=0;

int      error=0;
int      default=0;
int      translate=1;
int      supportUndo=1;
int      showModified=1;
string   oldValue;
int      align=alNone;
int      style=0;
int      taborder;
int      tag;
int      helpContext;
void     @value;
void     @items;
void     &onEnter;

void     &onExit;

void     @onChange;
void     @onClick;
void     @onHelp;
cblForm @panels[ ];
CMenu   popupMenu;
}

```

type	defines the type of control (see next sections).
caption	defines the caption of a control.
hint	defines the quick-Help text associated with this control.
enabled	(true=1, false=0) defines if this control can be used in the current context. Default is true. <code>enabled</code> is available for all controls. Disabled controls are marked as usual under Windows (grayed). No user interaction is possible with disabled controls.
readonly	(true=1, false=0) defines, if the control's value can be changed at runtime by the user. Default is false. Readonly edit fields get a gray background.
required	(true=1, false=0) defines if the control has to be assigned some value. Use <code>cblForm's CheckComplete()</code> to check the values of required parameters of a form. Default is false. Labels for required elements are shown in boldface.

<code>translate</code>	(<code>true=1</code> , <code>false=0</code>) activates the automatic translation for the current control. Default value is <code>true</code> (see <i>"CBL - National Language Support" on page 174</i>).
<code>visible</code>	(<code>true=1</code> , <code>false=0</code>) defines, if the control is visible at runtime. Default is <code>true</code> . Available for all controls.
<code>modified</code>	<p>(<code>true=1</code>, <code>false=0</code>) is set immediately by the CBL system when the user modifies the value of a control. If the user changes the value of a control back to its original value, the <code>modified</code> attribute is reset immediately.</p> <p>The <code>modified</code> attribute is also reset by a call to the <code>ResetModified()</code> method of the form. Usually you call <code>ResetModified()</code> after a call to <code>form.Get()</code>, when you have verified that the value of each control is acceptable and processing of the data starts. The <code>modified</code> attribute must not be manipulated directly by the CBL application!</p> <p>The visual effect is, that a change bar appears to the left of the <code>modified</code> control. The color of the change bar is set in the options dialog of the CBL Interpreter and is yellow by default.</p>
<code>outofdate</code>	<p>(<code>true=1</code>, <code>false=0</code>) defines, if the value of a control is not current. This attribute should be set by the CBL application, if you know that the value of a control is out of date, but you don't want to update it automatically, e.g. to avoid unnecessary data transfer to and from Hicom. Default is <code>false</code>.</p> <p>The visual effect is, that a change bar appears to the left of the <code>outofdate</code> control. The color of the change bar is set in the options dialog of the CBL Interpreter and is brown by default.</p>
<code>error</code>	<p>(<code>true=1</code>, <code>false=0</code>) defines, if the value of the control is erroneous. This attribute is set by the <code>CheckComplete()</code> method of a form, if a control which has the <code>required</code> attribute set, does not contain a value. The <code>error</code> attribute can also be set by the CBL-application. Default is <code>false</code>.</p> <p>The visual effect is, that a red change bar appears to the left of the erroneous controls (only if change bars are activated).</p>
<code>showModified</code>	Activates display of a change bar for this control. Default value is <code>true</code> . The change bar is only activated if displaying change bars for the current form is activated.
<code>supportUndo</code>	(<code>true=1</code> , <code>false=0</code>) defines whether the context-sensitive menu (right mouse key) for this control should include the menu item "Undo".
<code>oldValue</code>	This string is used to save the previous value (<code>value</code>) of a control element. <code>oldValue</code> can be set explicitly or via the functions <code>ResetModified</code> or <code>ResetModifiedControl</code> . In the case of the latter functions, <code>oldValue</code> is set to the current value. <code>oldValue</code> is also the value that is entered for the control on executing the Undo function. Furthermore, as soon as <code>value</code> and <code>oldValue</code> differ, i.e., <code>modified</code> is not equal to 0, the change bar is displayed.
<code>col</code>	column number which the control is assigned to.
<code>row</code>	row number which the control is assigned to.

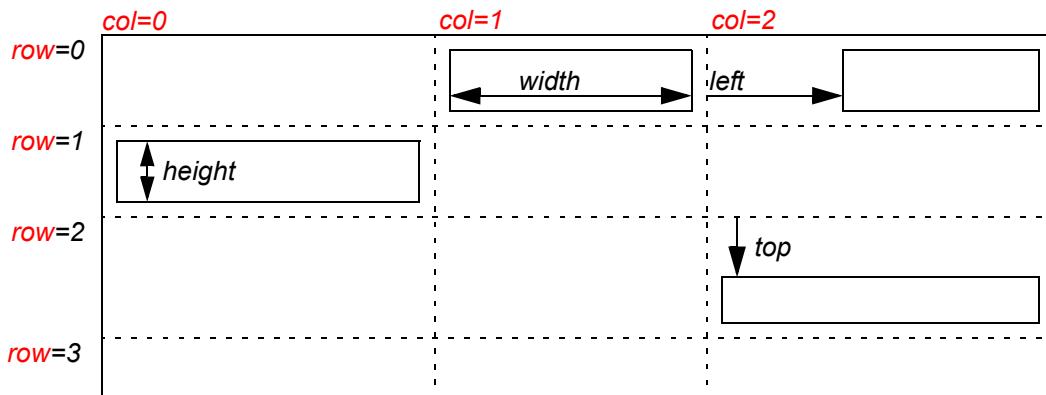
<code>colspan</code>	colspan and rowspan define the width and heights of the control in grid columns/rows. Default value is 1.
<code>rowspan</code>	
<code>left</code>	offset of the left position relative to the layout cell; in pixels.
<code>top</code>	offset of the top position, relative to the layout cell; in pixels.
<code>width</code>	width of the control in pixels.
<code>height</code>	height of the control in pixels.

`align`

This attribute is used to specify the alignment of a control (A) within the layout cell **or** (B) within the surrounding panel/form. Alignment with the surrounding panel/form (B) is used to define the layout of the control, when the window is resized by the user.

(A) Alignment relative to the current layout cell:

All controls can be assigned to an "invisible" layout grid by setting the attributes `col` and `row`. The width of a column of this layout grid is defined by the widest control in this column, the height of a row of the layout grid is defined by the tallest control in this row.



For controls with a caption the width of the control can be computed automatically by the "layout" (e.g. Label, Button, Checkbox, Radiogroup). The height of the control can be computed by the number of entries (e.g. Radiogroup). For controls that can contain other controls (e.g. panel) the size of this control is defined by the sum of the sizes of the contained controls.

The position of the control within a cell can be controlled by the value of the attribute `align`. In this case `align` can be a sum of values of `alHLeft` (adjust left, default), `alHCenter` (adjust horizontally centered), `alHRight` (adjust right), `alVTop` (adjust top, default), `alVCenter` (adjust vertically centered), `alVBottom` (adjust vertical bottom).

Example: `align:alHCenter+alVBottom`.

The size of a control within a cell can be defined by `align` too. To use the available horizontal space set `align` to `alHFill`, to use the available vertical space set `align` to `alVFill`.

Example: `align: alVCenter+alHFill`.

The size of a control and of the cell rows and columns is computed, when `form.Show()` is called and is not recomputed, when the window containing the controls is resized by the user!

`align`
(cont.)

(B) Alignment within the surrounding panel:

`align` can take the values `alNone`, `alTop`, `alBottom`, `alLeft`, `alRight` and `alClient`. Default value is `alNone`.

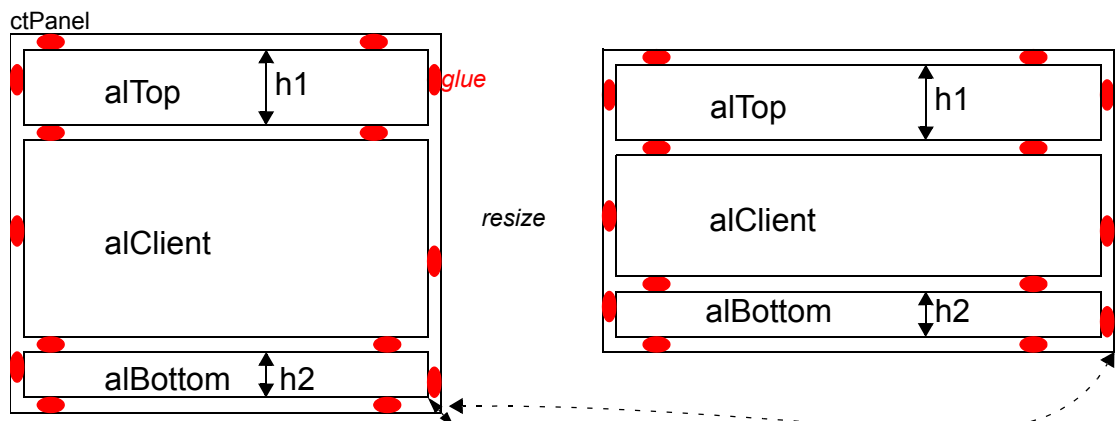
With this means you can divide a (sub-)form either horizontally or vertically into up to three areas: top - client - bottom, left - client - right. You may only use a combination of either `alTop`, `alBottom` and `alClient` or `alLeft`, `alRight` and `alClient`.

On each (sub-)form no two controls must have the same value other than `alNone` in their `align` attribute.

Setting alignment to another value than `alNone` means that the size of this control is defined by the surrounding panel or form, e. g. like an editor area (=memo field) that resizes with the surrounding form.

If `align` is `alTop` or `alBottom` only the height attribute of the control is considered. `top` and `left` are ignored, `width` is taken from the surrounding panel.

`alClient` makes a control a client control: both position and size is defined by the available space.



Controls which can contain other controls are `ctPanel`, `ctPages` and `ctGrid`.

see *"Alignment to resizable panels"* on page 171

<code>taborder</code>	<p>defines the order in which the controls are selected if the user moves from control to control using the tab-key. <code>taborder</code> is optional and should start with 1. The default tab order is defined by the order in the <code>controls[]</code> array which differs if the controls array is integer or string indexed (alphabetically sorted by the indexes' values).</p> <p>You may use the same value for <code>taborder</code> both for a control and its associated label. This is also useful to define a keyboard shortcut for a control: When the user enters the keyboard shortcut of the label, the associated controls gets the focus.</p>
<code>tag</code>	<p>can be used for general purposes by the user. It is only evaluated if a Button is clicked. Then the <code>tag</code> value is written into the <code>value</code> variable of the Button (if available). <code>tag</code> also defines the result value of the function <code>ShowModal()</code>.</p>
<code>helpContext</code>	<p>If F1 is pressed or the menu item "help" is selected from the context sensitive menu this value is passed to the <code>onHelp</code>-function of the form as argument.</p>
<code>value</code>	<p>is a reference to a variable holding the value for the current control. When the form's <code>Get()</code> method is called, the current values of all controls are copied from the form into the referenced variables. To avoid multiple write access to a variable, each variable must only be referenced in one <code>value</code> attribute within a form. Else the multiple write accesses to the variable would cause unpredictable results.</p> <p>However, there is one exception to this rule: The <code>value</code> attribute of several buttons should reference the same variable! After clicking one of the buttons you can determine which one was pressed by checking which <code>tag</code> value is found in the variable referenced by the <code>value</code> attribute.</p>
<code>items</code>	<p>is a reference to a variable holding an array of items for the current control (e. g. items of a combo box).</p>
<code>onChange</code>	<p>is a reference to a function which is called if the value of the current control is changed.</p>
<code>onClick</code>	<p>is a reference to a function which is called if the current control is clicked.</p>
<code>onHelp</code>	<p>is a reference to a function which is called if the menu item "Help" is selected from the context sensitive menu (right mouse key) of a control.</p>
<code>onNavigate</code>	<p>is a reference to a function which is called if the menu item "Go to object" is selected from the context sensitive menu of a control.</p>
<code>panels</code>	<p>is an array of subforms for compound controls (e. g. page control).</p>

`popupMenu` `popupMenu` allows to define a context sensitive menu for this specific control. The `onClick` handlers of the menu items get the reference of the current control as first parameter. Example:

```
void abcClicked(CFormControl @control)
{
    comWinFrame.ConsolePrint("ABC clicked");
}
```

`onEnter` is a reference to a function which is called if the current control gets the focus.

`onExit` is a reference to a function which is called if the current control loses the focus.

Please note that the event handling routines cannot change the value of the current control.

4.4.3.1 ctLabel

<code>caption</code>	<code>string</code>	label text
<code>value</code> *	<code>string @</code>	reference to a string variable containing the label text
<code>onClick</code>	<code>function @</code>	function which is to be called if label is clicked; a label where <code>onClick</code> is specified, is displayed with underlining like a HTML hyperlink.

* Either `value` or `caption` should be set. If `caption` is set, the label will only be updated if the form is repainted using `Show()`. If the label's text is stored in a variable a reference to this variable must be assigned to `label.value`. Then the label's text is updated with every call of `FormUpdate` (better performance).

4.4.3.2 ctEdit

<code>value</code>	<code>string @</code>	reference to a string variable containing the value of the edit field
<code>onChange</code>	<code>function @</code>	function which is called for every change in the edit field (each typed character!)
<code>style</code>	0	standard edit field
	1	edit field for password input (prompt character is *)

4.4.3.3 ctCombo

value	string @	reference to a string variable containing the value of the combo edit field
items	string [] @	reference to an array of strings to be displayed in the combo box
onChange	function @	reference to a function which is called for every change in the edit field (each typed character!) in the combo box.
onClick	function @	reference to a function which is called before the selection dialog is shown when the "..."-button is clicked. This function can prepare the items array. Only available for style=1.
style	0	standard combo box (Windows style)
	1	Enhanced combo box, consists of edit field and "..."-button to start a selection dialog. The selection dialog contains a string grid (do not confuse this with the ctGrid control!) which is filled with the items array. A ' ' -characters in each item separates the columns in this string grid. If the 'Ok'-Button is pressed the value of the first column is displayed in the edit field after the selection dialog is closed.
	2	standard combo box (same as style=0) but without the possibility to edit a value. Only selection of predefined values is possible (drop down list).

4.4.3.4 ctMemo

items	string [] @	reference to an array of strings to be displayed in the memo
-------	----------------	--------------------------------------------------------------

The memo control is readonly.

4.4.3.5 ctListbox

value	int @	reference to an int variable containing the index of the selected item (or -1 if none is selected)
items	string [] @	reference to array of strings to be displayed in the listbox
style	0	standard listbox (Windows style)
	1	Enhanced listbox. The array of items is displayed in as a table. ' ' -characters in an item separate the columns in this table.

caption	string	used to define column headers for listbox of style 1; ignored for listbox of style 0. ' '-characters in caption are used to separate the column headers.
---------	--------	-------------------------------------------------------------------------------------------------------------------------------------------------------------

4.4.3.6 ctCheckListbox

value	int [] @	reference to an int array containing the values (states) of the CheckListbox entries. The size must be the equal to the numbers of items
items	string [] @	reference to array of strings to be displayed in the checklist box

4.4.3.7 ctRadiogroup

value	int @	reference to an int variable containing the index of the selected item (starting with 0)
items	string [] @	reference to array of captions of the radio buttons
caption	string	caption of the radio group (displayed in the radio group frame)

4.4.3.8 ctCheckbox

caption	string	caption of the checkbox
value	int 0 or 1 or 2	0 if the checkbox is not checked, 1 if checked, 2 if grayed.
style	0 1	Checkbox can have the states checked and not checked Checkbox can be grayed, too.

4.4.3.9 ctButton

caption	string	caption of the button
onClick	function @	function to be called if button is pressed. If it returns an int result, the result is considered in deciding whether to close a modal dialog. The return value is only evaluated, if tag!=0 and style==0.
tag	int	return value for ShowModal(), if onClick EventHandler does not return an int result. Also result value for value
value	int @	gets the value of tag if the button is pressed (before execution of onClick)
style	0 1	if tag>0 a modal dialog containing this button is closed. even if tag>0, a modal dialog will not be closed.
default	int	if default!=0 this button is the default button for the current form.

4.4.3.10 ctAmoParameter

The AmoParameter control allows to define a HiPath 4000 Expert Access-MML-Editor styled control. All information about the AMO parameter is stored in the object that is passed to `item`. This object can be easily retrieved from HiPath 4000 Expert Access-MML-Editor using the method `getParamValidation(string param)`.

Example:

```
amoParam gerkon;
...
comedit.command="EINR-SCSU";
gerkon=comedit.getParamValidation("GERKON");
```

value	string @	gets the value of the parameter
item	amoParam	if default!=0 this button is the default button for the current form.

4.4.3.11 ctBitmap

caption	string	path to the bitmap to be displayed
---------	--------	------------------------------------

4.4.3.12 ctPanel

panels	cblForm[]	panels[0] contains the form which should be displayed within the panel.
--------	------------	-------------------------------------------------------------------------

The caption of *the form* defines the caption of the panel.

- If this `caption` is not empty, then `style` is ignored and a standard panel (Windows style) with a title in a frame is displayed.
- If this `caption` is empty the panel is just a rectangle with bevels defined in `style`.

style	0	the panel is a raised area (like a button)
	1	the panel is plane
	2	the panel is a lowered area
	3	standard panel (Windows style)
align		if align is <code>alLeft</code> , <code>alRight</code> , <code>alTop</code> , <code>alBottom</code> or <code>alClient</code> the attribute <code>style</code> is ignored. Instead the panel looks like <code>style==1</code> , if align is <code>alLeft</code> , <code>alRight</code> , <code>alTop</code> or <code>alBottom</code> and looks like <code>style==2</code> , if align is <code>alClient</code> .

4.4.3.13 ctPages

value	int @	current page index starting with 0
onChange	function @	function which is called when another page was selected by the user. This function should not take too much CPU time, because this function delays the switch to the next tab sheet.

<code>panels</code>	<code>cblForm[]</code>	array of forms which should be displayed in the different tab sheets. The caption of the tab sheet is the caption of the "sub-form".
---------------------	------------------------	--------------------------------------------------------------------------------------------------------------------------------------

4.4.3.14 **ctGrid**

<code>items</code>		<code>items</code> is the reference to an array of an user defined grid class which contains the actual values of the grid controls.
<code>panels</code>	<code>cblForm[]</code>	<code>panels[0]</code> contains the form which defines the controls to be used within one line of the grid.
<code>value</code>	<code>int var</code>	index of the actual grid line. This value represents the line index of the grid if an event handler has been called.

Grid controls are the most complex controls which can be defined on the GUI of CBL applications. The following example should show the usage of a grid control.

```
include "form.cbl"
```

```
class cblGridItem
{
    string caption;
    string value;
    string items[];
}
```

```
class gridItemCls
{
    cblGridItem A_edit1;
    cblGridItem B_combol;
    cblGridItem C_button1;
}
```

```
gridItemCls gridItems[];

gridItems={ { A_edit1:{value:"edit1"},
              B_combol:{value:"combo",items>{"v1","v2","v3"}},
              C_button1:{caption:"b1"} },
             { A_edit1:{value:"l2"},
               B_combol:{value:"c2"},
               C_button1:{caption:"b2"} },
             { A_edit1:{value:"e3"},
               B_combol:{value:"c3",items>{"a1","a2","a3"}},
               C_button1:{caption:"b3"} },
             { A_edit1:{value:"e4"},
               B_combol:{value:"c4"},
               C_button1:{caption:"b4"} }
             };

class helloCls
{
    int gridLine;
    cblForm form1;
    int x;

    void get()
    {
        form1.Get();
    }

    void update()
```

```
{
    form1.Update();
}

void ButtonClicked()
{
    form1.Get();
    string line;
    line=gridLine;
    gridItems[gridLine].A_edit1.value="Line "+line;
    form1.Update();
}

void init()
{
    form1 = { caption:"Grid Test Form",
              controls:
                { "grid1":
                  { type : ctGrid,
                    width :300,          height:90,
                    items : gridItems,
                    value : gridLine,
                    panels: {{ controls:
                                { "A_edit1": { type: ctEdit,
caption:"feld1" },
                                "B_combol": { type: ctCombo,
caption:"feld2" },
                                "C_button1":{ type: ctButton, value:x,
caption:"action", onClick:ButtonClicked }
                                }
                              }
                  }
                }
    }
```



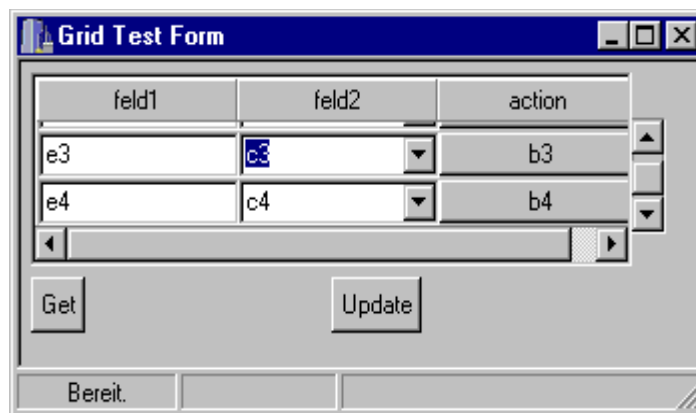
```

        }}
    },
    "but1": { type:ctButton, row:1, caption:"Get",
onClick:get },
    "but2": { type:ctButton, row:1, left:150,
caption:"Update", onClick:update }
}
};

form1.Show();
}
}
helloCls hello;
hello.init();

```

The result window looks like this:



4.4.4 Definition of menus

caption	string	is the text displayed as menu item and as hint for the speed button.
onClick	function @	is the function to be called if the current menu item is clicked.
submenu	cblMenu	is the submenu of a menu item.
enabled	0=false, 1=true	defines if the current menu item can be selected or not. Default is true.
left	int	defines the horizontal distance to the previous speedbutton in pixel. Default is 0.
bitmap	string	defines the file path to the bitmap which is to be used for displaying the speedbutton.

```
class cblMenu
{
    cblMenuItem items[];
}
class cblMenuItem
{
    string    caption;
    void      @onClick;
    CMenu     submenu;    // for menu only
    int       enabled=1;
    int       visible=1;
    int       left;       // for speedbar only
    string     bitmap;    // for speedbar only
}
```

4.4.5 Form examples

4.4.5.1 Alignment to layout cells

This example shows a form using many features of the previous described form description method.

```
// FormExample

include "form.cbl"

class myWindow
{
    CForm form1;
    CForm subform1;
    CForm subform2;
    CForm subform3;

    string label1="Label 1";
    string val1="Value 1";
    string val2="Val";
    string val3="Value 3";
    string val4="Choice 2";
    int val5=1;
    int val6;
    int val7;
    int val8;
    string[] choiceItems={ "Choice 1", "Choice 2", "Choice 3" };
    string[] choiceItems2={ "Choice 1|Description 1",
                           "Choice 2|Description 2",
                           "Choice 3|Description 3" };

    int pageIndex;
    void update() { form1.Update(); }
    void start() { form1.Get(); }
    void get() { form1.Get(); }
    void close() { form1.Close(); }
    void init()
    {
        subform1={ caption: "Page 1",
                   controls: { { type:ctLabel, col:0, row:0, caption:"label1", onClick:update },
                               { type:ctLabel, col:0, row:1, caption:"label2" },
                               { type:ctLabel, col:0, row:3, caption:"label3" } } };
        subform2={ caption: "Page 2",
                   controls: { { type:ctLabel, caption:"subform with
caption" } } };
        subform3={ controls: { { type:ctLabel, caption:"subform w/o
caption" } } };
        form1={ caption: "Hello World",
                mainmenu: { items: { { caption: "&File",
                                     submenu: { items: { { caption:"&Open", onClick:start },
                                                           { caption:"&Close", enabled:0 },
                                                           { caption:"-"},
                                                           { caption:"&Exit" }
                                                         } } },
                                     { caption: "&Edit",
                                       submenu: { items: { { caption:"&Cut" },
                                                           { caption:"&Paste" }
                                                         } } },
                                     { caption: "&Help" }
                                   }
                },
                controls: { "label1":{ type: ctLabel, col:0, row:0, caption : "Label"},
                            "label2":{ type: ctLabel, col:1, row:0, caption : "Label with onClick",
                                       onClick:close },
                            "label3":{ type: ctLabel, col:0, row:1, caption : "Edit"},
                            "edit1": { type: ctEdit , col:1, row:1, value : val1, width:80,height:25},
                            "label4":{ type: ctLabel, col:0, row:2, caption : "Combo, default"},
                            "combo1":{ type: ctCombo, col:1, row:2, value : val2, items: choiceItems,
                                       width:80,height:25 },
                            "label5":{ type: ctLabel, col:0, row:3, caption : "Combo, style=1"},
                            "combo2":{ type: ctCombo, col:1, row:3, value : val3, items:choiceItems2,
                                       style:1, width:80,height:25 },
                            "radio1":{ type: ctRadiogroup, col:2, row:4, caption:"Radiogroup",
                                       value:val5, items: choiceItems },
                }
        }
    }
}
```

CBL - Programmer's Reference

Designing forms

```

"label6":{ type: ctLabel, col:0, row:4, caption : "Listbox, default"},
"listbox1":{ type: ctListbox, col:1, row:4, value:val7,
             height:45, items: choiceItems },
"label7":{ type: ctLabel, col:0, row:5, caption : "Listbox, style=1"},
"listbox2":{ type: ctListbox, col:1, row:5, value:val8,items: choiceItems2,
             height:70, style:1 },
"check1":{ type: ctCheckbox, col:2, row:1,caption:"Checkbox1",value:val6 },
"pages1":{ type: ctPages, col:0, row:6, value: pageIndex,
           panels : { subform1, subform2 }, width:150, height:100 },
"panel1":{ type: ctPanel, col:1, row:6, style:0, panels: { subform1 } },
"panel2":{ type: ctPanel, col:0, row:7, style:1, panels: { subform3 } },
"panel3":{ type: ctPanel, col:1, row:7, style:2, panels: { subform3 } },
"panel4":{ type: ctPanel, col:2, row:7, style:3,
           panels: { subform3 } },
panels: {
  subform3
};
"button1":{ type: ctButton, col:0, row:8, caption: "Ok",onClick : start,
            enabled:1, tag:1 },
"button2":{ type: ctButton, col:1, row:8, caption: "Get",onClick:get },
"button3":{ type: ctButton, col:2, row:8, caption: "Update",onClick:update}
});
form1.Show();
}
}
myWindow window;
window.init();

```

This is the resulting form:

The screenshot shows a CBL form titled "Hello World" with a menu bar (File, Edit, Help). The form contains the following elements:

- Label:** "Label with onClick" (green text).
- Edit:** A text field containing "Value 1" and a checkbox labeled "Checkbox1".
- Combo, default:** A dropdown menu showing "Val".
- Combo, style=1:** A text field containing "Value 3" with a small button to the right.
- Listbox, default:** A listbox containing "Choice 1", "Choice 2", and "Choice 3".
- Listbox, style=1:** A table with two columns: "Choice" and "Description".

Choice 1	Description 1
Choice 2	Description 2
Choice 3	Description 3
- Radiogroup:** A group of three radio buttons labeled "Choice 1", "Choice 2", and "Choice 3". "Choice 2" is selected.
- Page Control:** Two tabs labeled "Page 1" and "Page 2". "Page 1" is active, showing a listbox with "label1", "label2", and "label3".
- Subforms:** Three rectangular boxes labeled "subform w/o caption".
- Buttons:** Three buttons labeled "Ok", "Get", and "Update".
- Status Bar:** A bar at the bottom with the text "Bereit.".



The "tagged" expression list has been used for initialization of the `controls` array of the form. If we do so, the controls array gets string indexed. Then we can access (and modify) each control element by its name, e.g. for activation of a button:

```
form1.controls["button1"].enabled=1;
```

4.4.5.2 Alignment to resizable panels

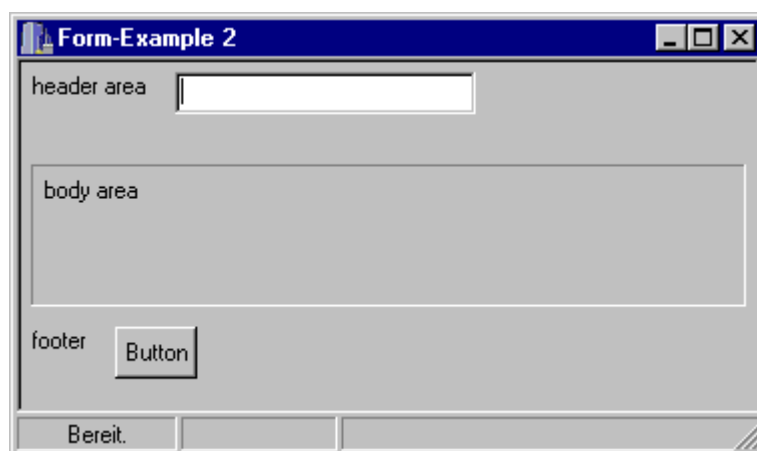
```
// FormExample2

include "form.cbl"

class myWindow
{
    CForm form1;
    CForm header;
    CForm body;
    CForm footer;

    void init()
    {
        header={ controls: { { type:ctLabel, caption:"header area" },
                             { type:ctEdit, col:1 } } };
        body= { controls: { { type:ctLabel, caption:"body area" } } };
        footer={ controls: { { type:ctLabel, caption:"footer" },
                             { type:ctButton, col:1, caption:"Button" } } };
        form1= { caption: "Form-Example 2",
                 controls: { "panel1":{ type: ctPanel, align:alTop, panels: { header } },
                             "panel2":{ type: ctPanel, align:alClient, panels: { body } },
                             "panel3":{ type: ctPanel, align:alBottom, panels: { footer } }
                           };
        form1.Show();
    }
}

myWindow window;
window.init();
```



4.4.6 Modal dialog example

This example shows a form using many features of the previous described form description method.

```
class COpenConnection_Dialog
{
    cblForm form1;
    string host;
    int show(string title)
    {
        CReply reply;
        int OK=1;
        int CANCEL=2;
        form1={ caption:title,
                controls: { { type: ctEdit,
                             caption: "IP address or\nlogical name :",
                             value:host,
                             layout:{left:30,top:10,width:100} },
                           { type: ctButton, caption: "OK", tag:OK,
                             layout:{left:70,top:60} },
                           { type: ctButton, caption: "Cancel", tag:CANCEL,
                             layout:{left:160,top:60} }
                         },
                layout:{left:250,top:200,width:290,height:150}
        };
        system.BringToFront();
        system.ConsoleClear();
        if (form1.ShowModal(this)==OK)
        {
            form1.Get();
            system.ConsolePrint("Initialisation of host "+host+" started.");
            reply=tools.open_conn(host,comwin);
            system.ConsolePrint(reply.lines[0]);
        }
        return(reply.status_ok);
    }
}
```

4.5 CBL - National Language Support

The goal of NLS (National Language Support) is to support writing language independent applications.

The CBL Interpreter supports two ways of defining national language depending texts. One HiPath 4000 Expert Access-own mechanism and one especially for HiPath 4000 Expert Access running in the H.O.T. (Hicom One Tool) environment.

Advanced topics

- > chapter 4.5.1, "CBL-intrinsic text tables"
- > chapter 4.5.2, "NLS texts from H.O.T."

4.5.1 CBL-intrinsic text tables

CBL allows the definition of text tables (see *"text_table_stmt" on page 16*).

This example shows the mechanism:

```

text_table      {   "de"           ,   "en"           ,   "es"           }
{
  "Bereit":      {   ""              ,   "Ready"        ,   "listo"        },
  "Läuft":       {   ""              ,   "Running"      ,   "elaboración"  },
  "Anzeigen":    {   ""              ,   "Show"         ,   "anuncio"      },
  "Abfragen":    {   ""              ,   "Query"        ,   "buscar"       },
  "Ändern":      {   ""              ,   "Modify"       ,   "modificar"    },
  "Löschen":     {   ""              ,   "Delete"       ,   "borrar"       }
};

```

Text tables can be used to define translations for language dependent strings. When the CBL program is read the global translation table is set up. Only the texts for *one* translation is actually stored by the interpreter even if there are more translations defined. The translation column is selected which corresponds to the value of a global variable named "Language". The initialization value is determined from the current user default language as stored in the Windows' registry database.

The tag (e.g. "Bereit:") is used within the application.

The translation table is used automatically for input and output texts of forms (e. g. label caption, combo box items etc.). The automatic translation can be switched off for each individual control by setting the attribute `translate` to 0 (=false).

If there is no value ("") defined as translation in the current language column, the tag value is used instead.

The translation table can be accessed by purpose using the interpreter defined functions "TranslateIn(string text)" and "TranslateOut(string text)".

Example: (Language=en)

TranslateOut("Anzeigen") returns "Show",

TranslateIn("Modify") returns "Ändern".

This list shows the abbreviations for the languages as supported in the current release of Hi-Path 4000 Expert Access:

german	de	english	en
spanish	es	dutch	nl
french	fr	finnish	fi
danish	da	portuguese	pt
korean	kr		

4.5.2 NLS texts from H.O.T.

NLS texts can be read at compile time from the HOT.NLS component.

| Please see "*syn_stmt*" on page 14 for details.

5 Introduction into CBL Programming

Advanced topics

- > chapter 5.1, "Preface"
- > chapter 5.2, "Data types, variables and operators"
- > chapter 5.3, "Statement and Expression"
- > chapter 5.4, "Object and Class"
- > chapter 5.5, "Programming with GUI-Builder"
- > chapter 5.6, "HiPath 4000 Expert Access Interoperation"

5.1 Preface

CBL is a special language which has been created to provide more possibilities to automate the commands or create the application rather than using only the batch command line to control the Hicom switch.

This "Step-by-Step to CBL" will be the introduction to some basic ideas in CBL programming to get start to program CBL with the basic knowledge and to be able to understand the CBL programming reference later on for creating the sophisticated program on demands.

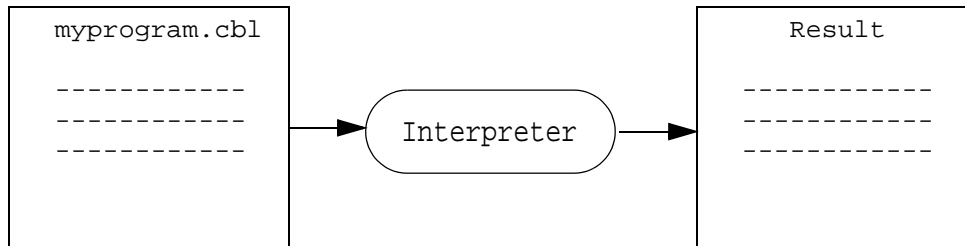
5.1.1 What is CBL?

CBL is the short form of Comtes Batch Language which was created to be the script language to communicate with all other components in the HiPath 4000 Expert Access platform. But it can also be considered as a general purpose programming language like C++, Basic or Java. CBL codes can be turned into the series of actions that do tasks or solve some problems which vary from the easy tasks (e.g. calculate some numbers) to the more specific and complicated applications that work between components (e.g. the program to show all the stations in Hicom switch).

CBL is relatively similar to Java or C++ in the way of syntax, object-oriented structure but even more simple to use. So it would be rather easy for the person who had already some experiences in programming and would not be too difficult for the beginner as well. In the programming structure aspect, CBL has the general style of primitive data types such as string, integer, array but can also define the complex data types by using classes. It can also be programmed in functions or methods which can be called later in the program. Besides the individual running program, it can be program to communicate with other HiPath 4000 Expert Access components such as HiPath 4000 Expert Access-MML-Editor or HiPath 4000 Expert Access-Frame by OLE-Automation.

5.1.2 What do we need to start?

Considering the type of CBL language, it is not a compiled language like C++ or Java which all the codes in the file will be read and compiled to generate the result by the compiler. On the other hand, CBL is an interpreted language like Basic or JavaScript which the code will be read and interpreted language expressions into actions statement by statement.



To start programming, we need to have an editor to write the code on. It works the same way as programmers who write Basic can use Visual Basic editor or Java use Jbuilder editor. Recently, CBL can be written on a very simple tool such as Notepad in Microsoft Windows operating system or any other kinds of editors. The CBL Interpreter and the HiPath 4000 Expert Access-GuiBuilder also has an built-in editor.

Currently ".cbl" files are associated with notepad by HiPath 4000 Expert Access installation.

Example: Hello world

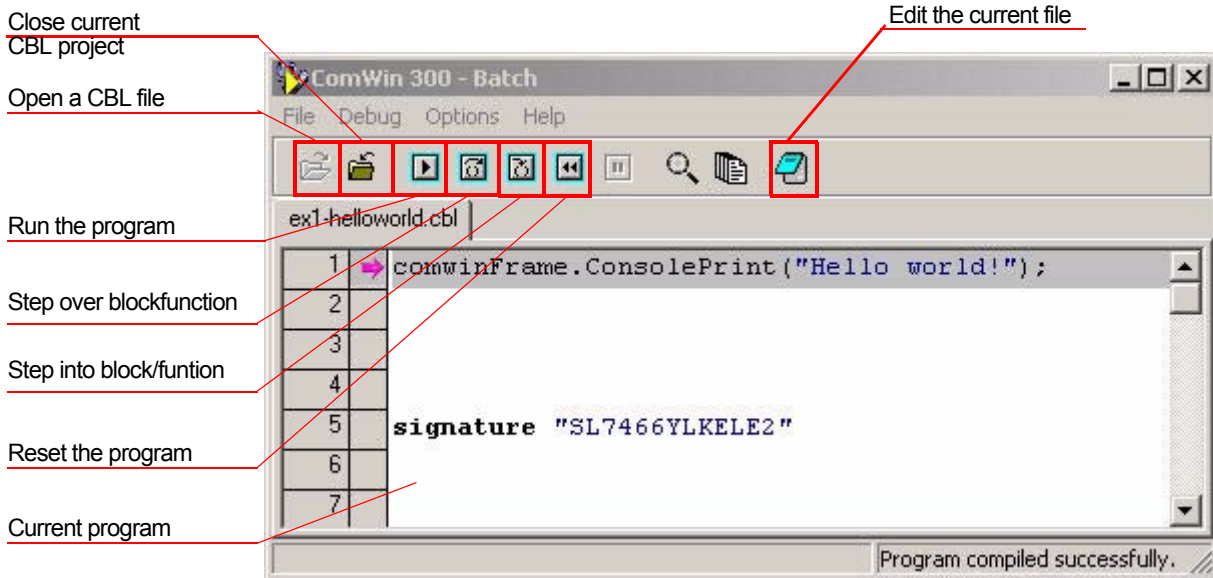
Start with the very famous simplest example, this example will give some ideas how the cbl program can be written, run and show the result under HiPath 4000 Expert Access environment. You can start now by following the steps as shown below.

1. Open some Editor tool (e.g. Notepad: You can find it in the Start program->Accessories->Notepad)
2. Then write the program to show the word "Hello world" by typing the code as followed.

```
comwinFrame.ConsolePrint("Hello world!");
```
3. That's all the code you have. Next, save it by pick the name of your file as you like but do not forget to put the extension file as ".cbl"
4. Now you have created the cbl file. To run it, first open the HiPath 4000 Expert Access CBL Interpreter. The interpreter "HiPath 4000 Expert Access - CBL Interpreter" looks like the picture below.

Introduction into CBL Programming

Preface



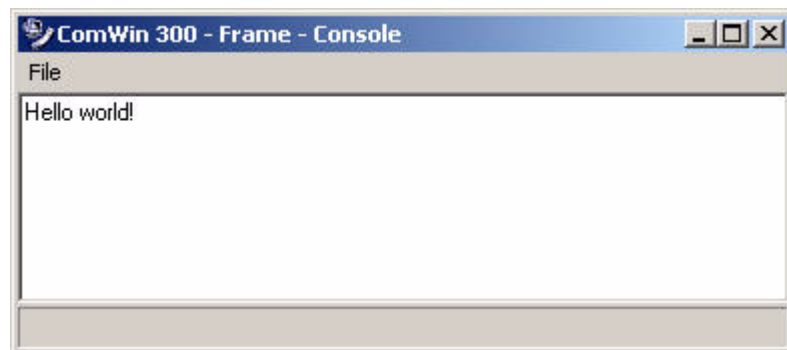
5. Load your cbl file into the interpreter by click Open icon (the first icon on the left) or select open in the menu File. Then you will have the file loaded. When the file is loaded the CBL syntax is checked.

Remark: The signature is automatically created after the file is loaded by the CBL interpreter. Currently it has no special function except it represents some internal checksum. In files you create you may remove the line but CBL interpreter will try to recreate it again. For the execution of the program it has no effect.

6. To run the program, press "Run" icon (the third button) in the menu bar. The code will be interpreted and the result will be generated. In this case, the word "Hello world!" will be shown in the HiPath 4000 Expert Access-Frame console window which will also be automatically popped up with the result inside.

HiPath 4000 Expert Access-Frame

Console window is now used as the output console which contains the text result from the program in example 1.



7. In case you want to edit the source code, it can be done both by press the notepad icon on the menu bar while the code is loaded or open the code file in Notepad in the normal way. After the code is edited, it must be saved first. Then load it in the interpreter again or if it was saved in the same name, just reload it by pressing "Reset the program" (4th icon in the menu bar) and run it again by pressing "Run program" button.

After grabbing the way to deal with the interpreter and the editor, we will go into more details in CBL language. Data types, program structure, and the more advance examples will be presented in the following chapters

5.2 Data types, variables and operators

Data types are considered as the first and the very essential step to master languages. Every language has data types as data abstractions or simply saying as a way to tell the compiler what kind of data using in the code in order to allocate the suitable memory, including the way to manipulate the data.

If you are familiar with the other programming languages, you might have known many data types already. Many languages may use different notions for the types but most of them have the same set of common basic data types eg. integer, string, float, boolean and etc. The same as CBL interpreter, it provides a set of predefined data types which composes of Integer, String and Array. It seems not so many data types predefined but it has been done on purpose. As from the objective of CBL which is mainly for switch controller, the data types offered as the primitive types would be enough to deal with this subject.

This chapter will present all primitive data types in CBL including the way to manipulate those data types with some examples and also about the variable.

5.2.1 Primitive data types

There are 3 data types which are included in CBL interpreter. In another word, it is called primitive data types or predefined types which the compiler provided with operators to manipulate. Integer, string and array are the 3 data types which have been mentioned. There are also another way to create complex data types by yourself which will be introduced later in the another chapter.

5.2.1.1 Integer type

Integer variables are stored as 32-bit numbers, which require 4 bytes of storage. Integers can be used to represent: the numbers in the designated range of -2147483648 to 2147483647. It can represent only the number without decimal or exponential multiplication. It can be used with 5, 96,324354, and so on within the range mentioned above but not 12.5, 0.214, 0.21×10^3 .

This numbers can be used in any other purposes in programming. For example it can be used to simulate boolean values which have two states: 1 for true and 0 for false or use as any identification numbers: the identification number of the stations.

Integer also has predefined operations which can be apply with this type of data, for example, interger number 3 and 5 had properties of integer so this number can be applied by the arithmetic operation like the basic operations such as plus(+), minus (-), multiply(x) and devide(/) and other operations as you can see in the CBL programmers guide/reference manual.

Introduction into CBL Programming

Data types, variables and operators

Example: Simple calculation

This example is a simple calculation of mobile phone bill for a month. This calculation will just show the idea how to manipulate with integer data in CBL language although it will not be the perfect calculation with 2 decimal digits like it is supposed to be in the real situation. In the integer system the value will be rounded (if it is not integer in result) before assigned to the integer type.

First, in this calculation, we have the monthly payment for the connection service, tax, call out cost for this month and we want to know the total cost altogether. Then start with all four values in the declaration part. We can also initialize the value of monthly payment right there by putting "=" followed by the amount. In this example, the monthly payment is around 13 Euro and tax is 16% so this program will use those numbers as the example. The name of the integer data type is `"int"`.

```
int monthly = 13;
int callcost;
int tax;
int total;
```

This following line is just an output display for the title of the program.

```
comwinFrame.ConsolePrint("This is my mobile phone cost calculation.");
```

Start the calculation process, from assigning "callcost" for this month, assume that it is 50 Euro. Then total price needed to pay will be callcost + monthly. After that, calculate tax which is 16% and include tax in the total price.

```
callcost = 50;
total = callcost + monthly;
tax = total*16/100;
total = total + tax;
```

Display the result of the calculation by having the lines below. For the syntax, in case of text output, put the text in "...", and if it is a variable, put it without "...".

```
comwinFrame.ConsolePrint("I have to pay");
comwinFrame.ConsolePrint(total);
comwinFrame.ConsolePrint("Euro!! for my mobile!!");
```

Like in the last example, the result will be shown in HiPath 4000 Expert Access - Frame - Console. The calculation of this program will come out with 73 Euro with all the texts as shown below.

```
This is my mobile phone cost calculation.  
I have to pay  
73  
Euro!! for my mobile!!
```

If you have some errors instead, check the spelling first. Then if you still cannot get it, may be you forgot the ";" at the end of the statements. Everybody will have simple errors at the beginning, so you donot have to worry about it. When you have more experience, errors will become less and you will enjoy programming.

5.2.1.2 String type

The second primitive type provided by CBL interpreter is "**string**". It is used to represent the characters or text. String has also some operations to manipulate with data. Some of them are common with integer type but some are special operations for string which cannot be applied with integer. For example, "in" operation cannot be used by integer data type and some integer operations cannot be applied with string type, eg. Minus(-) operation.

The frequent use operations for string type are concatenation(+), "in" operation, assign operation(=).

Example: String concatenation

In this example includes the string type assignment and concatenation. In the editor first we declare the variable a, b and c to be string type by put the name of the variable followed by type. Then we will use a, b and c for some basic operations for string.

```
string a;  
string b;  
string c;
```

These two lines are for string assignment. We assign the text to the variable a and b.

```
a = "Hello world!!";  
b = "Today is Friday, Yeah!";
```

For the variable c, we assign it's text to have text a, text b and more additional sentence. We can do it by using (+) operation which will concatenate the string in the variable a, b and the string c together. So the new text which will be assigned to c will be "Hello world!!" + " Today is Friday, Yeah!" + "Let's go out..."

```
c = a + b + "Let's go out...";
```

This comming last line in the program will print out the string in variable c.

Introduction into CBL Programming

Data types, variables and operators

```
comwinFrame.ConsolePrint(c);
```

The result will come out as followed.

```
Hello world!! Today is Friday, Yeah! Let's go out...
```

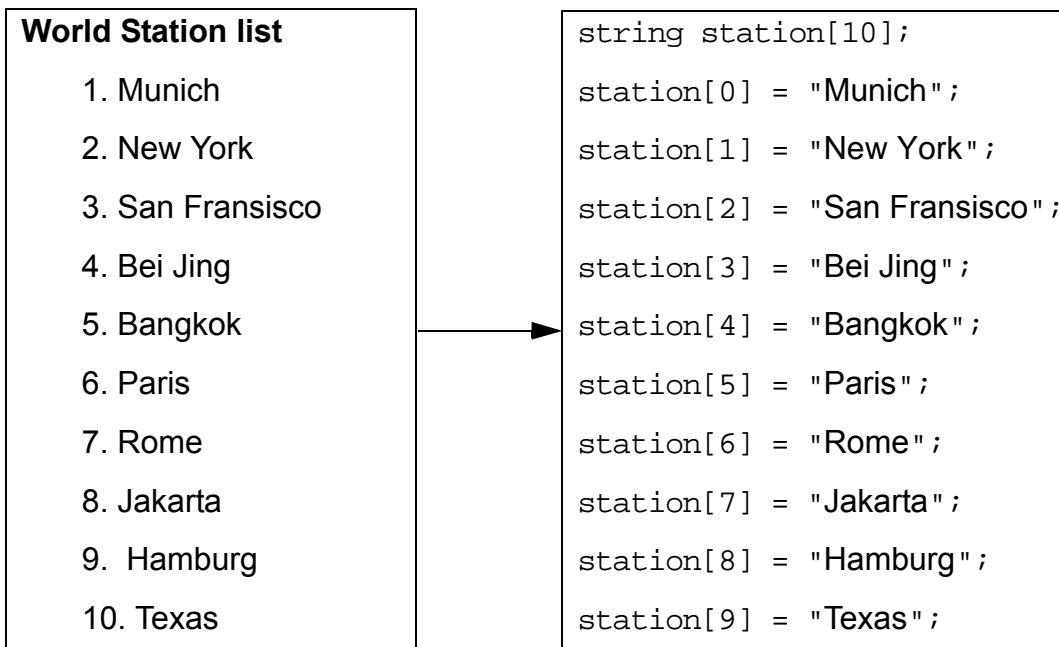
For other operations of string, e.g. "in" operation can be used to check whether the specific text is a part of another text or not. As stated in the sample code below.

```
string text1;  
...  
if ("no" in text1)  
{subtask1}  
...
```

From the program, if there is the text "no" occurred inside the text in variable text1, subtask1 will be executed. Another usecase of "in" operation will be with String still have many other useful operations which can be found in the handbook for referencing.

5.2.1.3 Array type

Array is another useful primitive data type. Like in other languages, array used to represent the set of variables which have similar type. Thus, it has to be used with another data type, e.g. array of string or array of integer. For example, we have 10 stations which located in the different places and we would like to have variables of those ten with their location. We can use array of string because those ten station's locations have the same data type which is string.



As from the diagram, the locations are store in the array of station which in this case "limited size array". There are two types of array, classified by the size. This first one is limited size array which programmer has to specify the size in the array declaration.

```
string day[7];
```

The variable "day" is a set of strings that contains only 7 of items as decleared. But in the case of unlimited size array, the array's items will not be limited. With this way studentname array can be added up unlimited as the example.

```
string studentname[];
```

The next important topic involved with array is indexing. As from the set structure of array, it has many items in the list. Thus, when the items need to be accessed, they need the indicators to identify themself which item are they. Each item in the list has its own identity idication which is an index. From the first example diagram, you can see the 10 stations which are having numbers from 0-9 in the bracket. Index numbers always start from 0, so 10 stations have index number 0-9. In the code, if you want to change the location of station 1 from Munich to Leipzig, Station 1 can be reached by mention Station[0].

```
station[0] = "Leipzig";
```

Type of the array just mentioned is *Integer indexed array* which is widely used in many languages.

In CBL, there is still one more type named *string indexed array*. This type, as the name, use string to be the index instead of integer which sometimes make more sense to do this way. For example, the case of employee's extension numbers, we can represent those data easily by using this way.

```
int[] x;  
x["Erich"] = 52142;  
x["Bernd"] = 56984;  
x["Robert"] = 54632;  
x["Emil"] = 58963;  
...
```

So we can just use names as the indices to access each item in array. For instance, to display the telephone extension of Erich can be done directly by using indexed name and display the value.

```
comwinFrame.ConsolePrint(x["Erich"]);
```

The string indexed array has also the integer index set which mapped from the string indices, ordered alphanumerical. Thus, it can also be accessed like in the normal integer index array by mention the integer index. From in the telephone extension above, we can order the indexed name respectively as Bernd, Emil, Erich and Robert.

Introduction into CBL Programming

Data types, variables and operators

x[0] is mapped to x["Bernd"]

x[1] is mapped to x["Emil"]

x[2] is mapped to x["Erich"]

x[3] is mapped to x["Robert"]

So these below statements mean the same thing.

```
comwinFrame.ConsolePrint(x["Erich"]);  
comwinFrame.ConsolePrint(x[2]);
```

For initializing value in array, in the integer indexed type can be initialized as follow,

```
string setstring[] = {"Erich", "Robert", "Bernd"};  
int setnumber[] = {1, 2, 3, 4};
```

On the other hand, the string indexed type can be initialized this way.

```
int extensionset[] = {"Bernd":56984, "Erich":52142, "Emil":58963}
```

Remark: Only items that exist in the array can be accessed, otherwise the interpreter will give a warning message. For example, a couple lines of code below declare an array which has two items.

```
int item[];  
item[0] = 32;  
item[1] = 16;  
comwinFrame.ConsolePrint(item[2]);
```

Although item[2] is not yet defined the above statement will work. If an array element of an integer indexed array is accessed that is not yet defined a new element with the default value (for integer this is 0) will be created automatically.

```
int[] x;  
x["Erich"] = 52142;  
x["Bernd"] = 56984;#  
comwinFrame.ConsolePrint(x[2]);  
comwinFrame.ConsolePrint(x["James"]);
```

The access with x[2] result in an error message because of range violation (only elements 0 til 1 defined) but with x["James"] interpreter will print the result of 0, represent the default value of item which is not exist (in case of item data type is string, it will show no character.)

5.2.1.4 Variables

After data types, we have been using variables already in many examples. Now this will be explained more in details. Start from the answer of the question "What is a variable?" It is obvious that "variable" is not a new word. At least people knew it from Mathematics. Infact, concept of variable in both fields are basically comparable. In the definition, a variable is a symbol or a name that represent its data value. In programming, having variable is providing more flexibility to have dynamic changing of data. A variable provide another alternative by assigned the value in real time rather than using only fix value in the program.

The definition might look a bit complicated but a variable can be simply explained as the place used to store the value which can be changed over time. For example, a mobile phone has some memory to store the phone number. You can put your friend's name and telephone numbers and they will be stored in the memory. Over time, a friend "John" moves to another city, so you get his new number then you can replace the new number within the same name. The old number which is stored in "John" changed to a new one. This is happened with computer memory and variable which its value can be changed overtime. We can make use of the this feature. Without it computer programming will not be considered of a very useful innovation.

To use variables, we need to provide the name of the variable and also the data type of the variable. These two are called identifier and type identifier.

5.2.1.5 Identifier and type identifier

Identifier is not only used as the name of the variable but also for funtions and the other user-defined items which can be defined in condition with some rules. Identifier can be characters which can start with a letter or an underscore and the following characters must be either letters, digits, or underscores as the example shown below.

Correct identifier	Incorrect identifier
employee	test-2 : It has "-" .
_throw	abc..d : It has "..".
book_1	12ab : It begins with digits.
salary12	_test : It is reserved for CBL system

Identifiers are also case sensitive e.g. the variable named a and A are treated as the different variable.

5.2.1.6 Type identifier

Type identifier defines the data type of a variable and return value of the funtion. For variables they can be specified by the type integer, string and they can also be array of those two types as well.

Introduction into CBL Programming

Data types, variables and operators

- Integer uses "int" as the type identifier.
- String uses "string" as the type identifier.

Both of them have to be in the lower case letters and follow by the variable's name.

5.2.1.7 Declaration

Declaration is one of programming statements. It is important to start using variables. Compiler / Interpreter will be informed about the variable from the declaration. It will allocate the memory space to the variables according to their data type.

A variable can be declared in two ways. First, it can be declared as the constant value. This case is useful when the program used many times a constant number. Thus, when you want to change this constant value, it is easier to change only one time the value of the variable instead of tracking all the codes to change line by line. Second, variable is declared as a dynamic value variable which we have seen already.

The syntax of the first case of constant variable starts with keyword "const" then followed with the type identifier then the identifier and assigned value, e.g.

```
const int montly_payment = 13;
```

Another case of normal variable, syntax would start with the type identifier then follows with the identifier.

Example: Integer declaration

```
int variable1;  
int set1;
```

Example: String declaration

```
string booklist;  
string text;
```

In case of array, the declaring variable's name has to be followed by brackets. As seen in the last topic of array, it has two different methods to define which are limited size array (with the upper bound) and unlimited size array (without limited bound). It has two different types of access which are string indexed and integer indexed. Interpreter will allocate the memory space to array only when a new array element is accessed but not when it is defined. Therefore, accessing an element which is not yet created causes an empty element.

```
int book[];  
int student[10];  
string test[];  
string text[3];
```

Scope of variables:

The next question might be "where can I put the declaration statement?". Variables can be declared in 4 different places which are :

- inside blocks (every sequence of statements that are enclosed by curly braces { } , e. g. inside or outside of a function)
- in the definition of function parameters
- inside class, but outside of functions
- globally: outside of all classes and functions

Variables stated in the different places also have different functions which are local variables, formal parameters, member variables, and global variables respectively.

Example: Variable's scope

```
int gMax=10;           // global  
int result;  
  
int add(int a, int b) // function parameters  
{  
    int sum;           // local scope (inside block)  
    sum = a+b;  
    return sum;  
}  
result = add(5,3);
```

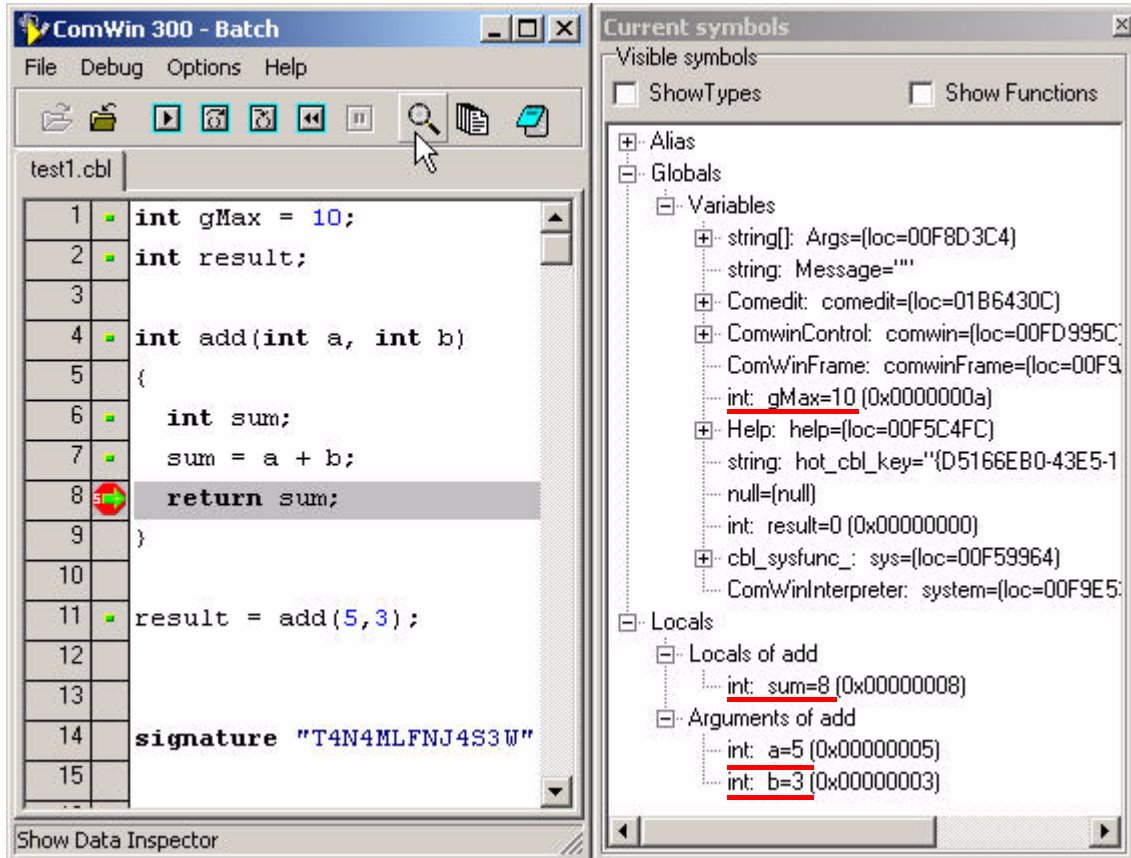
Global variables and local variables have some differences. The global variable declared outside the function and it can be accessed throughout the entire program but the local one can just be used only inside the local scope which it has been declared to be used in. For example, gMax is a global variable and it can be used even inside the function add but the variable sum can be used only inside the function block that it was declared in.

In HiPath 4000 Expert Access - CBL - interpreter, there is a feature called data inspector which allowed you to trace the value of all variables including the scope information so that you can also see the global and local variable scope. To use data inspector, click an icon on the menu bar (the third one from the right). It will open the data inspector window (the window with the

Introduction into CBL Programming

Data types, variables and operators

title"Current symbols"). Like in the picture below, at the break point in function add, there are an amount of variables active at that moment. In the window, it shows values of interesting 4 variables which has underlined.



5.2.2 Operators

After got some ideas about the variables, then what can we do with them? To manipulate variables we need operators. In previous topics in this chapter, we have seen some operators for fulfilling explanation of data types, e.g. some arithmetic operators: plus (+), minus(-) which are arithmetic operators or assignment operators but there are still more useful operators in CBL language. Operators can be classified into many types and have different functionalities. Here CBL operations will be explained according to type of the operations.

5.2.2.1 Boolean operators

Before getting into boolean operators, it's necessary to know about boolean concept. Boolean value can be either true or false. It is always used in programming. Sometimes you want the program to do something if a specific *condition* you want is true, but do something else if it is false.

For example, if there is a person name "John" in the Hicom switch, read his extension number and output it on the screen. So if computer finds "John" in the switch the condition will become "true" and computer will proceed the task of reading and showing John's extension number. Otherwise, it will not do any task. That 's why boolean operators are very useful.

In CBL, there are two boolean operators given: logical and, logical or. Besides logical operators there are still be bitwise operators which are also using somehow the same logic to apply with operands. From now each operation will be explained one by one.

Logical AND (&&)

This operator has two inputs and will give the result in one output. The following example will show some ideas of this operation.

Example:

We have 3 expressions to consider.

1. "John needs a mobile phone"
2. 2. "John has enough money"
3. 3. Will John buy a mobile phone?

So for the AND operation, if expression 1 and 2 is true that John needs a mobile phone and also has enough money, he will buy a mobile phone. Those sentences can be represent using "&&" as a symbol for AND operation.

```
JohnWillBuy = (John has enough money)&&(John needs a mobile phone)
```

With AND, if any of the expression is false, it will result in false. So in these cases of :

- John has not enough money but he needs a mobile phone. He will not buy it.
- John has enough money but he does not need a mobile phone. He will not buy it.
- John has not enough money. He does not need a mobile phone. He will not buy it.

The only case that made John buy a mobile is both are true.

Thus, the logic of AND operation can be present as in table.

Operand1	Operand2	Result
False	False	False
False	True	False
True	False	False
True	True	True

Introduction into CBL Programming

Data types, variables and operators

Logical OR : (| |)

OR operation use the operator symbol, | | . It works in the similar way like AND operator but the only different is the logic.

Example:

The given 3 expressions are:

1. "The bin is full."
2. "Today is Saturday."
3. Jim will throw away garbages?

The expression will be

```
ThrowGarbages = (The bin is full) | | (Today is Saturday)
```

From this expression, Jim will throw away garbages in condition if the bin is full or it is Saturday. For any of the true input expression will cause the true result. But the fault case will be only if both inputs are false. The bin is not full and today is not Saturday. So Jim will not throw away garbages.

You can also see from the logic table of OR

Operand1	Operand2	Result
False	False	False
False	True	True
True	False	True
True	True	True

5.2.2.2 Arithmetic operators

All arithmetic operators can be applied with integer except addition operation which will turn to be string concatenation if apply with string. The function of each operator are already known in mathematics.

Operator	Symbol	Example
addition/ string concatenation	+	int a = 7+2 string text = "I know " + "CBL language"
subtraction	-	int b = 8 - 4;
multiplication	*	int c = 4*5;
division	/	int g = 10/5;
modulus	%	int r = 10%3;

5.2.2.3 Bitwise operators

Bitwise operators can be used to operate with integer variables in their binary data representation. There are a number of bitwise operators.

Operator	Symbol	Example
AND	&	Operand1 & Operand2
OR		Operand1 Operand2
XOR	^	Operand1 ^ Operand2

You might still have a question of how is each operator work? Now each bitwise operator is going to be explained and also with some examples.

Bitwise AND (&)

Every bitwise operators, including AND operator, will take the integer and change to binary number as digital bit representation. Then it will compare both operand bit by bit. The result will be according to AND logical as explained already in Boolean operation. Only 1,1 will make the result become 1, the rest will be 0.

```
int operand1 = 2;
int operand2 = 3
int result;

result = operand1&operand2;
comwinFrame.ConsolePrint(result);
```

The bit representation of the operand 1,2 will become like showing below and also the result.

Bitwise AND	Decimal value	Binary bit represented value
Operand1	2	0 0 0 0 0 1 0
Operand2	3	0 0 0 0 0 1 1
result	2	0 0 0 0 0 1 0

From the example, the last bit on the right of each operand will be compared and "AND" operation applied. So the result of 0,1 will be 0. The next bit of 1,1 will be 1. Then 00000010 is equal to 2 in decimal number.

Bitwise OR (|)

The way of proceeding in this operation is the same as bitwise AND but just change the logic applied with two operands. In OR operation if any of the compared bits have 1 then the result will become 1, the only case that will cause 0 in result is when both compared bit are 0,0.

Introduction into CBL Programming

Data types, variables and operators

Bitwisw OR	Decimal value	Binary bit represented value
Operand1	2	0 0 0 0 0 0 1 0
Operand2	3	0 0 0 0 0 0 1 1
result	3	0 0 0 0 0 0 1 1

In the same codes as the last example with bitwise AND, the result will just change. As from the last bits 1,0 result in 1. So the result will become 00000011 which is 3 in decimal number.

Bitwise XOR (^)

The logic of XOR is not the same as AND, OR. It take two base two values to compare then if they are the same, the result will be 0 but if they are different, the result will be 1. All possibilities in XOR logic can be shown below.

Operand1	Operand2	Result
0	0	0
0	1	1
1	0	1
1	1	0

Then using another sample codes for this operation. We have the codes like given:

```
int Operand1 = 90;
int Operand2 = 3;
int result;
result = Operand1^Operand2;
comwinFrame.ConsolePrint(result);
```

As always, the interpreter will interpret integer and take it as binary to do the operation which can be shown in the table below.

Bitwise XOR	Decimal value	Binary bit represented value
Operand1	90	0 1 0 1 1 0 1 0
Operand2	3	0 0 0 0 0 0 1 1
result	89	0 1 0 1 1 0 0 1

Then we will show the logic dertermination of this example bit by bit, starting from the first bit on the right to the left.

```
0 ^ 1 = 1
1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 0 = 0
```

So the final result will be 01011001.

5.2.2.4 Equality operators

Equality operators can be very useful when you want to have the conditions like: less than, equal to, greater than. Then you code your conditions easily by using these operators.

For example, if you want to say " apply 16% tax just for the employees who has more than 3000 salary. But the rest who has below 3000 will have only 10% tax" It can be present by pseudo-code(not the real code but more english related just to make it easy to understand) like this.

```
if (salary > 3000)then
    taxrate = 16%
otherwise
    taxrate = 10%
```

So it is obvious that without the equality operators, it will not be easy to express those expressions. Thus, CBL provide also equality operators which use also symbol to represent those meaning of "less then", "greater than" and so on. You can see all of them in the table below.

Operators	Symbol	Example	Result
equal	==	4==4	True
not equal	!=	5 != 6	True
greater than or equal	>=	5 >= 9	False
less than or equal	<=	6 <= 3	False
greater than	>	9 > 6	True
less than	<	7 < 3	False

5.2.2.5 Assignment operators

Before comming to this issue, you have been using already the assignment operators. The symbol of this operator is "=". In other examples, we have seen a lot of assignment process, such as:

```
int x;
int y;
string text;
x = 0;
```

Introduction into CBL Programming

Data types, variables and operators

```
y = 10;
text = "hello!";
comwinFrame.ConsolePrint(x);
```

You can see the assignment expressions in the example, e.g. the line of `x =`, `y =` but what does it mean? The process of assigning is very simple as it takes value of the expression on the right-hand-side and assign it to be the value of the variable in the left-hand-side. So `x` will have 0 as its value, `y` will have 10, `text` will become "hello!".

Assigning integer value at the declaration statement, we can put "=" followed by integer number. With the string, we use the same "=" but follow with the string in double quotes sign, such as:

```
int a = 5;
string text = "Hello world";
```

There are also other operators in this assignment group which are very useful and make the command shorter. Assume that you want the codes below:

```
int x;

int a;
x = x+a;
```

You can write the code in another way around by using the operator `+=` like this:

```
x +=a;
```

This line means the operator takes the value on the left-hand-side and add with the value on the right-hand-side then assign to the variable to the left-hand-side. In this case, the value of `x` will be taken and add by the value of `a` then assign it to the variable `x`;

There are some more operators with the arithmetic operators mixed with the assignment operators and it works in the same way as `+=`. They are all including `+=`, `-=`, `*=`, `/=`, `%=`. They will do the same method but change only the arithmetic operators.

Assignment as the type conversion

Here comes another issue in assignment. Let's look at the expressions below.

```
int x = "Try";
string y = 1234;
```

Those expressions are correct or not? You might think, they are all wrong. Some will say they are all type mismatching. Although it is quite strange to have the string assign to the integer variable and also on the other way around with integer value assign to string variable but in

CBL, it is allowed to do so. But why and how? It is just an implicit way to convert the data to be in the type of the variable. Because of the data from Hicom switch (both numbers and text) normally comes in text form. So if you want to manipulate the data which is in text format e.g. you have "12345" and you want to do some arithmetic operations with it. You have to change it to integer somehow. Then it can be applied by the operators. And also on another way around to change the integer to string to send to Hicom switch.

Example:

```
int aNum;  
aNum = "254";
```

From the codes above, the string "254" is converted to be integer and then assigned to aNum.

```
string sTxt;  
sTxt = 245;
```

As in the same way, the integer 245 will be converted into string "245"

But what will happen in these following cases.

```
int a = "test";  
int b = "12 34";  
int c = "58p";
```

In case the text assigned to an integer variable cannot be converted to number like the case variable a, it will just assign value 0 to a. But in case of b, the text has two groups of numbers separated with the space. So this case it will take the first value of the text which is 12. For the c case, the string will be converted to 58 because it cut out the character "p" behind.

Thus, besides the normal way to use assignment operators, it can also be used for type conversion purpose. This mechanism also works in another way not really at the declaration part, eg:

```
string y;  
int x;  
x = "123";  
y = 123;
```

Those lines can result also in the same implicit type conversion.

5.2.2.6 Operation precedence

Interpreter takes the different priority of manipulating data due to the operation precedence. As you can see from the example:

```
x = -3
x == -3 || x >= 0
x * -23 + 44 / 2
x = 5 / 2
```

x will be 2, because all numeric variables are integer (the remainder will be truncated)

```
x = 5 % 2
```

x will be 1, the remainder of the integer division. The semantics of modulo is the same as in C.

```
hello = Text
```

The precedence of all operators is:

Highest	.
	[
	(
	* / % in
	+ -
	< <= > >=
	== !=
	&
	^
	&&
	!
Lowest	

The precedence might look difficult to remember but to learn it by heart is not the good way. There is also another way.

```
x = 3-5*2; // This might cause a wrong precedence determination
x = (3-5)* 2 // It is clear that "-" is the first then "*".
```

Thus, the easier way is using parenthesis to specify the order of operations yourself.

5.3 Statement and Expression

Last chapter we dealt with the primitive data types and how to use them and also looked at the way to use and declare the variables with types. The last thing we did is operators. This chapter will be involved with how we organize what we have in chapter 2 to code the whole program with conditions and direct how the program supposed to be executed.

5.3.1 Expression

We have seen already many expressions in the last chapter. So what is exactly an expression? An expression is as simple as it contains one or more operations which represent by operators. The expression in the simplest way can be only the variable or number or string value such as:

```
MonthlyPayment  
123  
"Hello!"
```

It also can contain an operator, such as:

```
123 + 234  
"Hello" + "world!"  
9 & 5
```

Besides the expression with one operator, if it has more than one operator, it will be called *compound expression*. In order to deal with the execution of many operators, it has to determine the precedence of each operators it has, e.g.:

```
3/5-2  
6 / 3 | 5 + 2
```

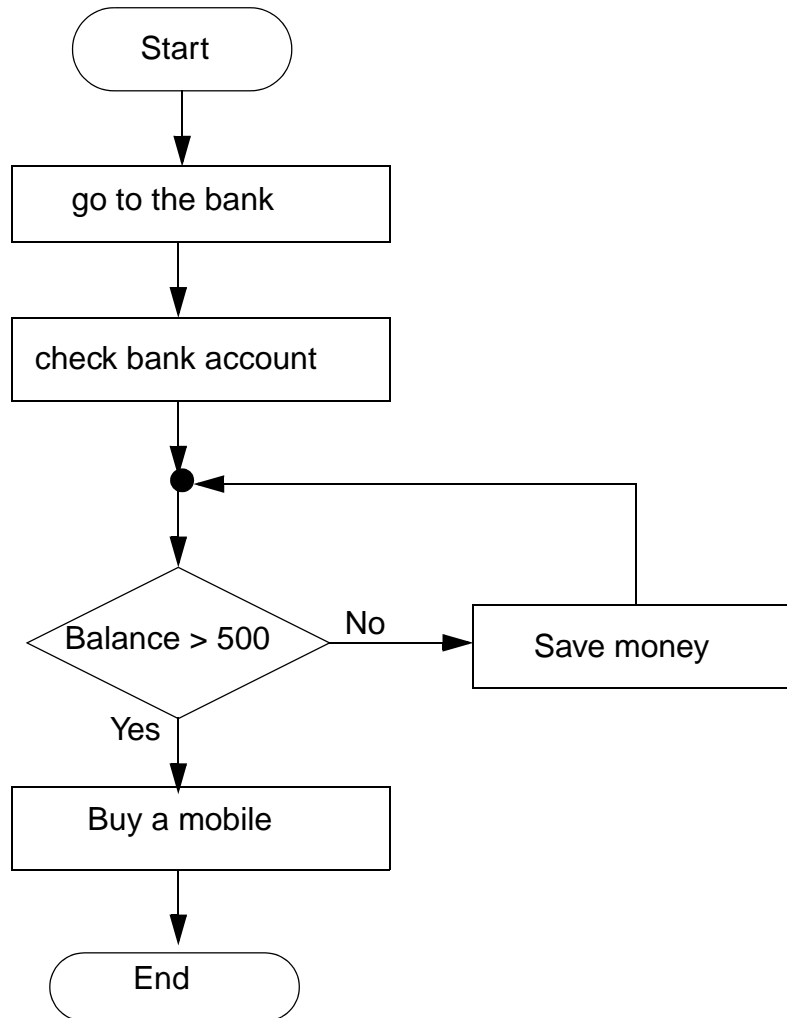
The operators in the expression can be any of the operators mentioned in chapter 2, e.g. arithmetic operators, boolean operators, bitwise operators and etc.

5.3.2 Statement flow control

The heart of programming is to execute commands automatically according to the purpose of the program. So the way to automate the commands without any help of human decision is to construct a flow control. A program can be broken up into parts which are control by a flow control.

Example: Flow control

John will buy a mobile phone if he has enough money. So he has to check his bank account first, and if he has more than 500 Euro, he will buy a mobile but if not, he will start saving money everyday until he has enough money to buy. Then he will buy it. The following diagram will show the flow chart of a program how it is separated into steps and the way to decide the branching decision point.



5.3.2.1 Function - statement

In order to make the program easier and understandable, the whole complicated program need to be seperated into many small parts which are much easier to handle and to code. Functions are the key to do this job. Each function represent series of tasks in one name which can be used (call function). Then in your program you can write functions and control the flow easier by calling functions in the main program flow.

There are two types of functions; first, functions without any return value and second, functions with a return value. When a functions with a retuen value is called, it will proceed its task then return a value as a result. This retuen value can be in any of the valid data types in CBL language. Function has a quite simple syntax which starts with an identifier to specify the type of the return value (in case the function has no return value, "void" will be used there). Then the function's name will be put. Another thing next to the name will be a parenthesis which can contain parameters to use inside the function or can be just an empty one.

Example: Function with a return value

```
int saveMoney(int amount)
{
    balance = balance + amount;
    return balance;
}
```

From John's example, saving money is a routine job which can be assigned as a function. This `saveMoney` function has an input parameter which is the amount of money to save. Then the saving amount will be added up with the old balance in the bank account then return the new balance as an integer value.

Example: Function without a return value

```
int x;
void myFunction()

    int v;
    v = 30;
    x = v + v/2;
}
```

This example shows how to use the none-return function. This function will not return any value but only proceed the tasks in the function. So when the function `myFunction` is called the `x` equation will be calculated and the `x` value will be assigned.

5.3.2.2 If - statement

The "if - statement" is a conditional statement which allows you to write the condition and to control the program to execute some codes due to the conditions. So if the *if-expression* is considered to be true (can be anything except 0) then the corresponding code block will be executed. Otherwise the block of *else* will be executed.

Example: If-else

```
x = 8;
if (x > 5)
{
    x = x * 10;
}
else
{
}
```

Introduction into CBL Programming

Statement and Expression

```
x = x * 20;  
}
```

In this example, first, x is assigned to be 8. Then it is checked in the expression (x > 5) which become true. Thus, the block next to the expression (x = x * 10) is executed. On the other hand, in case the value of x makes the expression become false (in case x < 5), the block after *else* (x = x * 20) will be executed.

If you want to make the conditions more complex, you can use the boolean operators, such as, and (&&), or (| |), not (!) and if you have more than one condition you can use *else-if expression* which will be checked one after the other if the previous condition is false.

Example: if - else if

```
int income;  
int tax;  
if (income < 500)  
{  
    tax = 0;  
}  
elseif ((income >= 500) && (income < 1000))  
{  
    tax = income * 10/100;  
}  
elseif ((income >= 1000) && (income < 2000))  
{  
    tax = income * 12/100;  
}  
else  
{  
    tax = income * 15/100;  
}
```

This example shows the simple process of tax calculation which the tax will be calculated from the income level. The income will be checked whether which level it is in from the first *if-expression* and go to the next *elseif* (if the condition is false) until the program found the range that the income is in. Then it will calculate the tax with the percentage of that income level.

5.3.2.3 for - statement

This statement used for repeating the execution the block of codes due to the conditions gave in *for-statement*. For the person who is familiar with pascal or C/C++ language already. CBL for-loop is really similar to them. The first style which is like C language will have a syntax com-

pose of 3 expression after "for" which will define the initial value of loop control variable, a relational expression that determines when the loop exists, and the condition that defines how the loop control variable changing in each round.

Example: For-loop in C style

```
for (x = 0; x < 10; x++)
{
    saving = saving + 2;
    comwinFrame.ConsolePrint("Today my saving is =");
    comwinFrame.ConsolePrint(saving);
}
```

This for-loop will increase the value of `saving` variable in each time the loop is repeated. The loop start with a control variable `x` assigned to 0 then increasing by one as from the last expression in the parenthesis. It will be repeated until the value of `x` cannot meet the condition `x<10` in the second expression.

The initialization part can be omitted if the initialization of the variable has been executed before the *for-statement*.

```
int x = 1;
for ( ;x<10;x++)
{
    comwinFrame.ConsolePrint(x);
}
```

But if all of the expressions in for-loop are empty, it means the loop will run forever. The code in the block will be executed again and again.

```
for ( ;; )
{
    comwinFrame.ConsolePrint("I'm running forever");
}
```

However, there is a `break`-statement which can terminate the loop from inside the block:

```
int x;
for ( ;; )
{
    x++;
    if (x>10)
```

Introduction into CBL Programming

Statement and Expression

```
        break;
    }
```

Example: For-loop in Pascal style

```
for i = 1 to 100 do
{
    comwinFrame.ConsolePrint("I can count number :");
    comwinFrame.ConsolePrint(i);
}
```

The syntax in pascal style using "for... to" and it is ended with "do " and the code block in the bracket. In this example, the loop will be repeated 100 times and show the text as the codes in the bracket will be repeated.

5.3.2.4 while - statement

Another possibility to use loops in your program is employing *while -statement*. There are two types of *while -statement* in CBL language. The *while* and *do while-statement* are slightly different in syntax and logic of the statement. While loop will be run only if the expression in condition is first proved to be true. On the other hand *do while-statement* will in anyway execute the code block first then check the condition in the while expression at the end of the code block. Thus it will be repeated until the expression is still true. The different is *do while-statement* will execute the code block once anyway before check through the condition at the end but *while -statement* will check the condition first, so it will not execute the code block even once if the condition is not true at the beginning.

Example: while loop

```
int x = 0;
while (x < 10)
{
    saving = saving +2;
    comwinFrame.ConsolePrint("Today my saving is");
    comwinFrame.ConsolePrint(saving);
    x++;
}
```

This example is using the first style of *while-statement*. The block code will be executed only if the condition $x < 10$ is true. Then the program will increase the value of saving variable and print out the result.

Example: Do while - loop

```
int x = 0;
do
{
    saving = saving +2;
    comwinFrame.ConsolePrint("Today my saving is");
    comwinFrame.ConsolePrint(saving);
    x++;
} while (x < 10)
```

This program will first run through the code block after *do* and then check the while expression whether the condition is true or not. If the $x < 10$ is true, then the code block will be repeated otherwise the loop will be ended.

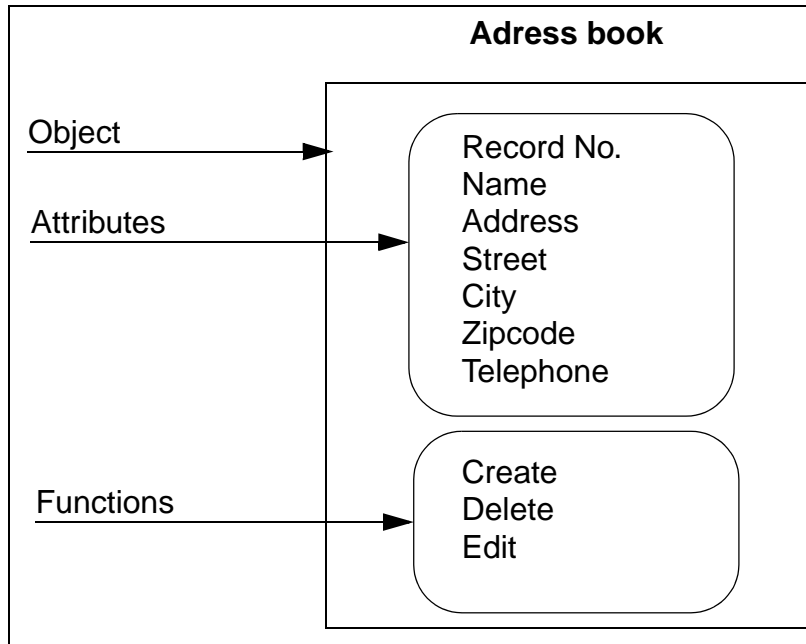
5.4 Object and Class

The new concept of Object-Oriented programming can be seen as an enhancement of the old style of procedural or structural programming. With the object oriented method, the program complexity will be reduced and will be more structured so that it made things easier for programmers. The advantages will be increasing the modularity, reusable features and more as you will see from the details later on.

5.4.1 What is an object?

To understand more about object-oriented programming, it is important to understand first the object concept. Object is everything around you or one can say anything in the universe, e.g. a car, a man and etc. With this concept, it is easier to model the complex things in real world to the abstract environment which can be used in programming. The key points of an object are its two types properties, data attribute and the function. Like the functions in the last chapters, an object can perform the tasks which are defined as its function. But unlike the a function, an object can do many predefined tasks and can also have its own data stored internally which is defined as its data attributes.

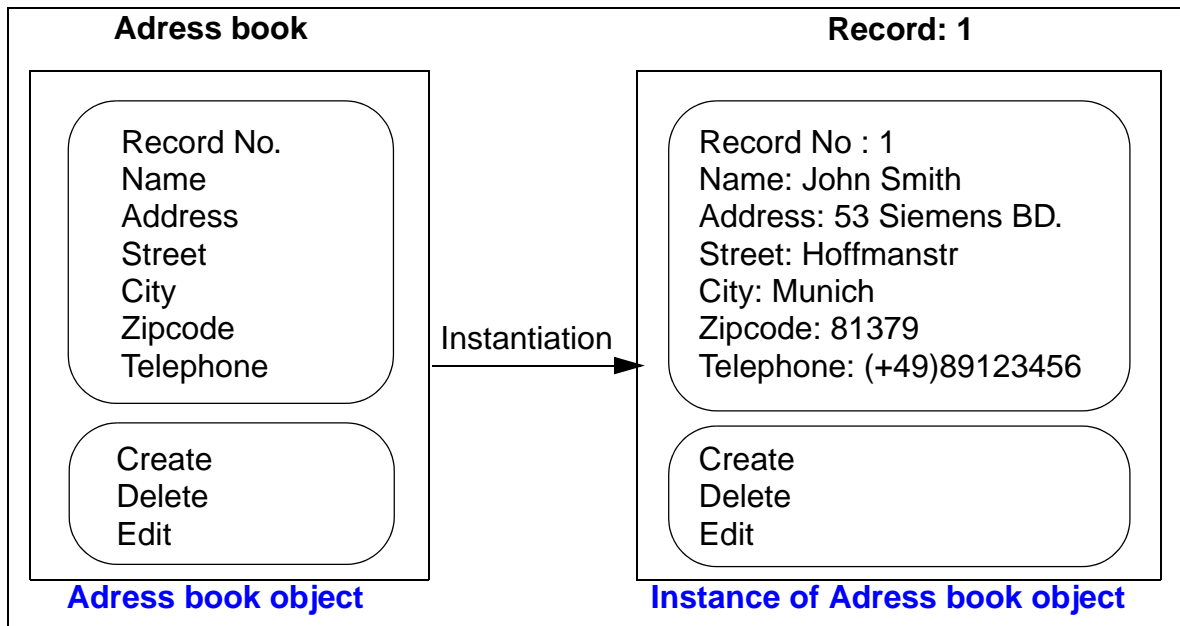
The example of the object can be an address book. The data attributes of this object can be the data field in the address books such as name, house number, street, city, zip code, telephone. For the actions, they can be all the actions that can apply with the address book: create, delete and edit a record. The object would look like this:



The figure above shows the address book as an object with the data attributes which is called *Member data (Attribute in CBL)* and the functions that applied with this object, called *Member functions*. Member data can be compared with the normal variables which explained before but it is different from the normal variables in the way of accessibility. Member data cannot be accessed everywhere like the global variable but can only be accessed internally by the object. Member functions also work in the same way as member data. It can be called only by referencing to the object.

5.4.2 Object instantiation

After applied the object oriented concept and designed the objects, they can not be directly used. This designed object is used as a block or a template for creating the manipulatable object which will be called *instances* from now on. For example, the address book is a template and the intances of this template are the data records you put in the address book:



This record is an instance of the address book. Each instance has its own internal data stored and it's own accessible functions. Every instance has the same pattern. It contains the same data members and the same functions from the template object.

5.4.3 Class

A template or the block that used to create the instances is actually called a class in programming language. As mentioned from Data type chapter that besides the primitive data types, there is another kind of data types. It is called complex data type which can be created on demand by using class representation. For instance, the address book can be assigned as a complex data type. Thus, the address book type will contain data members, e.g. record number, name, address and etc which can be assigned with primitive types: record number can be integer; name, address, city can be string. Besides the attributes, the address book type will contain also the functions for manipulating instances. The functions; delete and edit; are also needed to be inside the class.

Example: Address book class

```
class address_book
{
    int recordno;
    string name;
    string address;
    string street;
    string city;
    string zipcode;
    string telephone;
    void displayPhone()
}
```

Introduction into CBL Programming

Object and Class

```
{
    comwinFrame.consolePrint(name
                                + "'s telephone number is "+telephone);
}
void delete() { /*..delete a specific record..*/}
void edit() { /*..edit the record..*/}
}
```

Like any other statements, class also has some syntax in CBL. The syntax looks not so new because it consists of known statements which can be either *function statements* or *declaration statements*. As in the example, the syntax of the class declaration start with the word `class` then the class name and the declaration of attributes and function members in the a curly bracket.

In order to use the class, you have to make the instance out of it then use the instance. The syntax of instantiation is as simple as declare a variable. On the other hand, you can take the class as a data type then assign it to a variable. For instance:

```
void main()
{
    address_book john;
    john.name = "John smith";
    john.telephone = "089-123456";
    .
    .
    .
    john.displayPhone();
}
```

This case the data members are initialized outside the class but it can be done also in the class so that the new instance will have the default value from right after it is created.

```
class point
{
    int x = 0;
    int y = 0;
    int z = 0;
}
point p1;
comwinFrame.consolePrint(p1.x);
comwinFrame.consolePrint(p1.y);
comwinFrame.consolePrint(p1.z);
```


In this class, the class represent point in 3 dimensional coordinate system which each point in x,y,z axis has been initialized to 0. Thus, all points that are created from this class will have x,y,z value of 0 as the default value.

The member function of classes is like the normal function. It can be both with the input parameters or not. In the last example of address book, there are only functions without parameters. The syntax of member functions exactly the same as the normal function as the functions without parameters will start with *void* and the name of the function then function's statements inside curly bracket. Here is another example of

```
class person
{
    string name;
    int age;
    string sex;
    void incrementAge()
    { age++;
      comwinFrame.consolePrint("Now the age is");
      comwinFrame.consolePrint(age);
    }
}
```

If we create a person from this class, this person will have the function `incrementAge` which can be call without any parameters. Here is some codes of that show the way to call this function.

```
person john;
john.name = "John Smith";
john.age = 25;
john.incrementAge();
```

The same way as using the attributes of the instance, to call a function (also called method in this context)function, we need to mention the instance first. Thus, if we declared the instance name "john" as a person, the attributes and functions inside can be refered by using the instance name and a dot then follow with the specific attribute or function. Calling functions just have a bit different compared with the attributes that it needs to have the parenthesis after functions' name. If a the member function has parameters, it is called needed to have the name of parameters inside as shown in the function is followed by a parenthesis. example below.

If the member function has parameters, it is needed to have the parameters inside as shown in the example below.

```
class person
{
    string name;
    int age;
```

Introduction into CBL Programming

Object and Class

```
string sex;
int incrementAge(int y)
{ age = age + y;
  return age;
}
```

Now in this example, the incrementAge function has a parameter which is the amount of year to be incremented to the current age. The function's type is not void but it is assigned to the type of return value which in this case "age" variable's type. So when you call this function, it will automatically output the age variable after the process inside the function, for instance:

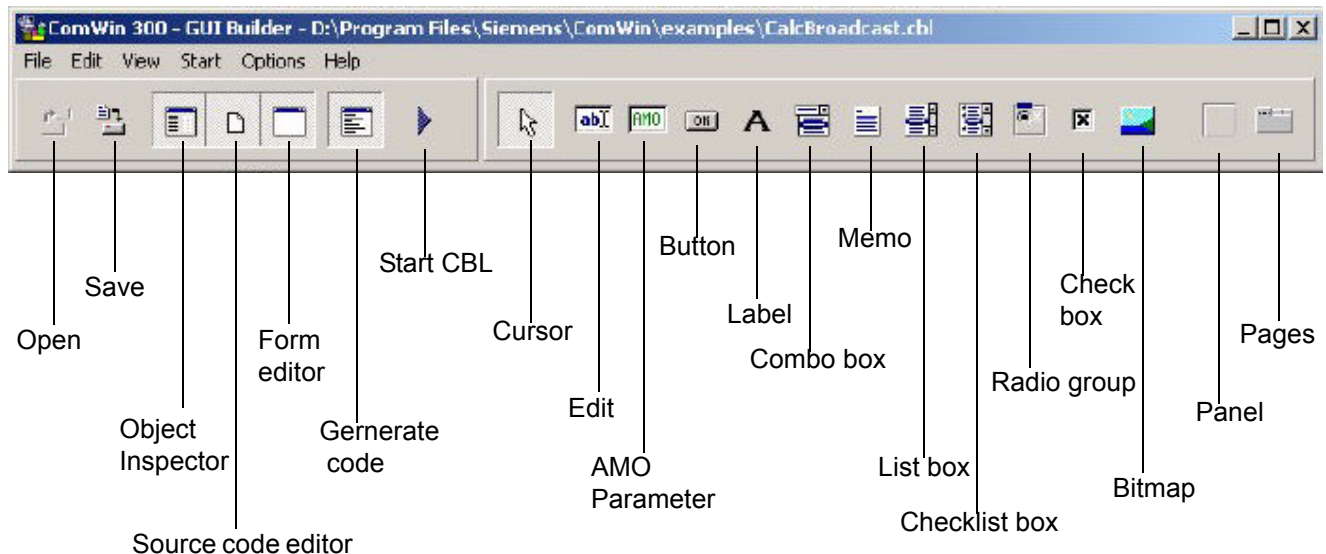
```
person john;
person jim;
int different;
john.age = 25;
jim.age = 20;
different = john.age-jim.age; // now the different will be 5.
john.age = jim.incrementAge(4); // But John& Jim is now 24.
```

The last expression assigns the return value of Jim's incremented age to John's age. So Jim's age will become 20+4 and then assigned to John's age.

5.5 Programming with GUI-Builder

Many programs have some interface to interact with the user. Thus, besides the program's mechanism or the functions of the program, interface design and implementation are also the important points that programmers need to know. The good Interfaces should be made to be easy to understand and to use so that in these days interfaces are generally designed in the graphical way with dialog boxes, buttons and input text boxes as we have seen in many programs in windows OS environment.

CBL language provide a so called "form library" that allowed you to use ready-made code to program your graphical interfaces easier but it is anyway not the best and comfortable way. Although interface coding with a support of library is possible, it is still be a hard work. HiPath 4000 Expert Access provide a tool which make the interface coding part easier. This nice tool called HiPath 4000 Expert Access CBL - GUI Builder which has quite similar features to C++ Builder interface or any other visual programming interface. It allowed you to create a form, buttons, edit boxes and etc. by using clickable menu bar.



GUI builder helps you to create your interface easily by providing menu bar to create forms, buttons, edit boxes. You can easily put the mouse cursor on the icon to see each icon's description.

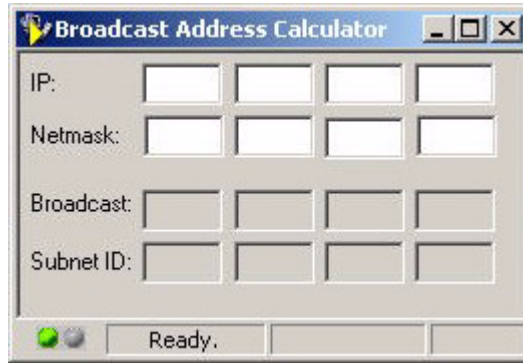
Example 5-2: Network broadcasting calculation

This example is a small program doing the calculation of network broadcasting address and subnet ID by using two inputs from the users which are IP address and the netmask. After having HiPath 4000 Expert Access - CBL- GUI Builder program on, you will see the program menu. Firstly, most often use buttons should be pressed on. They are the 3rd, 4th, 5th, 6th which will show the Object inspector, source code editor, form editor and generate code..

Introduction into CBL Programming

Programming with GUI-Builder

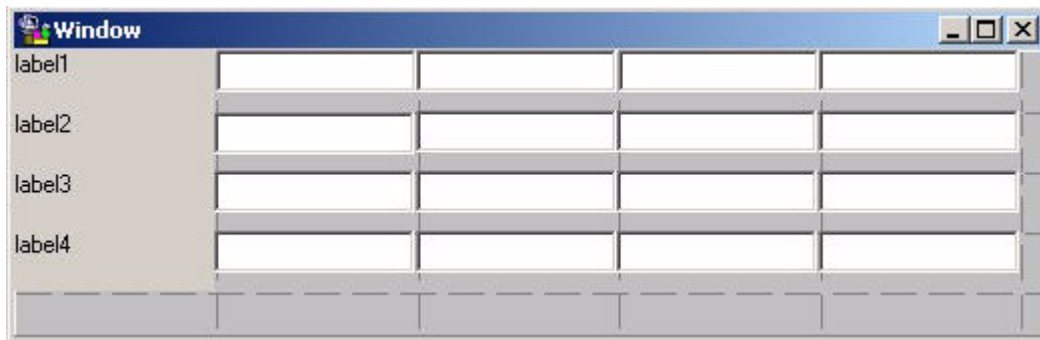
The interface of Broadcast address calculation program look like the picture on the right with two sets of IP address and netmask for inputs and two sets of outputs in the grey boxes which are Broadcasting address and Subnet ID. The output will be calculated on every change of the input fields.



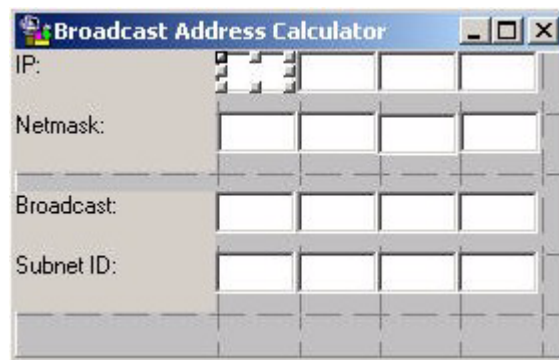
1. The step of creating interface would rather be easy for this program.
 - Create new project by going to menu program File-> New. Then the new form, Source code editor and object inspector will be popped up.
 - You can see and config object properties in the object inspector. For instance, the empty form now named Window you can change at the caption to make it "Broadcast Address Calculator" also all other properties like can also be changed.
 - It needs 4 labels for IP, Netmask, Broadcast and Subnet ID. This can be done by click at the label icon then click at the form to put the label on the right position.
 - Each line need 4 edit boxes to put the 4 parts of IP address. It can be done in the same way as the label but select the icon edit instead.
 - To make it easier to follow and read the code, it is better to name the object in the same name as we do. So in the object inspector, some of the object name and value needed to be change
 - Form change the name from Window to Main, caption to Broadcast Addressxs Calculator
 - Edit boxes named "editIp1" to 4 from left to right and Value is editIp1Value to 4 also
Name: editNetmask1 (to 4) Value: editNetmask1Value (to 4)
 editBroadcast1 (to 4) editBroadcast1Value (to 4)
 editSubnetId1 (to 4) editSubnetId1Value (to 4)

The two output lines which are Broadcast and Subnet ID should be set up to be read-only which can be done by checking Read-only checkbox in each box's property.

Remark: The status panel (which is showing "Ready." in the picture above) and the green and red(now grey) status lights are automatically done by the program. You do not have to create it yourself.



From the first step of layout the interface as the picture above, you can change the properties of the object including size of the edit boxes, title of the form and every captions to be your choices. Then it's ready to be programed.



- Coming to the programming part, once again, the goal is to calculate the broadcasting address and the subnet ID in every change of each field in IP or Netmask. So we need all the input edit boxes has to provide some be put the code for each event handler for on the onChange event of the input fields. If you select the first IP box, there will be the Events part on the bottom of object inspector. Double click in the onChange box, a function will be created. Then you have to put the code inside each function.
- For the code, 8 functions are more or less using the same algorithm. The only minor points are changed. This part of the code will look like this.

```
void Show()
{
    init();
    MainForm.Show();
}

// Function editIp1OnChange()
void editIp1OnChange()
{
    MainForm.Get();
    editBroadcast1Value=_string( _int(editIp1Value)
        | (0xff ^ _int(editNetmask1Value)) );
    editSubnetId1Value =_string( _int(editIp1Value) & _int(editNetmask1Value) );
    MainForm.Update();
}
```

Introduction into CBL Programming

Programming with GUI-Builder

```
}

// Function editIp2OnChange()
void editIp2OnChange()
{
    MainForm.Get();
    editBroadcast2Value=_string( _int(editIp2Value)
        | (0xff ^ _int(editNetmask2Value)) );
    editSubnetId2Value =_string( _int(editIp2Value) & _int(editNetmask2Value) );
    MainForm.Update();
}

// Function editIp3OnChange()
void editIp3OnChange()
{
    MainForm.Get();
    editBroadcast3Value=_string( _int(editIp3Value)
        | (0xff ^ _int(editNetmask3Value)) );
    editSubnetId3Value =_string( _int(editIp3Value) & _int(editNetmask3Value) );
    MainForm.Update();
}

// Function editIp4OnChange()
void editIp4OnChange()
{
    MainForm.Get();
    editBroadcast4Value=_string( _int(editIp4Value)
        | (0xff ^ _int(editNetmask4Value)) );
    editSubnetId4Value =_string( _int(editIp4Value) & _int(editNetmask4Value) );
    MainForm.Update();
}
```

Similar to the changes of the IP address input fields the onChange event handlers for the net-mask input fields needs to be provided:

```
// Function editNetmask1OnChange()
void editNetmask1OnChange()
{
    MainForm.Get();
    editBroadcast1Value=_string( _int(editIp1Value)
        | (0xff ^ _int(editNetmask1Value)) );
    editSubnetId1Value =_string( _int(editIp1Value) & _int(editNetmask1Value) );
    MainForm.Update();
}
```

```
// Function editNetmask2OnChange()  
void editNetmask2OnChange()  
{  
    MainForm.Get();  
    editBroadcast2Value=_string( _int(editIp2Value)  
        | (0xff ^ _int(editNetmask2Value)) );  
    editSubnetId2Value =_string( _int(editIp2Value) & _int(editNetmask2Value) );  
    MainForm.Update();  
}  
  
// Function editNetmask3OnChange()  
void editNetmask3OnChange()  
{  
    MainForm.Get();  
    editBroadcast3Value=_string( _int(editIp3Value)  
        | (0xff ^ _int(editNetmask3Value)) );  
    editSubnetId3Value =_string( _int(editIp3Value) & _int(editNetmask3Value) );  
    MainForm.Update();  
}  
  
// Function editNetmask4OnChange()  
void editNetmask4OnChange()  
{  
    MainForm.Get();  
    editBroadcast4Value=_string( _int(editIp4Value)  
        | (0xff ^ _int(editNetmask4Value)) );  
    editSubnetId4Value =_string( _int(editIp4Value) & _int(editNetmask4Value) );  
    MainForm.Update();  
}
```

4. Then the program can be run by clicking the "Start CBL" button on the menu bar. It will ask you to save the project first then it will be run.

5.6 HiPath 4000 Expert Access Interoperation

HiPath 4000 Expert Access architecture composed of many components which are able to communicate to each other and make the complicated program easier by using many components.

For a detailed description of these components and their interfaces please see *"Interfaces to the HiPath 4000 Expert AccessFrame components" on page 96*

Example: Display a list of station numbers

```
// Display all SCSU and SBCSU station numbers with PEN
// Some ComWin-CBL-Include files are automatically included (e.g.
// comwin.cbl) which contains the interface to ComWin-Connect and
// ComWin-Edit. sysfunc.cbl defines a lot
// of very useful function for processing of MML commands.
// OpenConnDlg defines an universal dialog for opening of Hicom
// connections which are stored in ComWin-Connects connection
// directory. If OpenConnDlg is used in the demonstrated way
// a CBL application can be started from a open connection via context
// menu Macro-/CBL-Bookmarks as well without change.

include "OpenConnDlg.cbl"

// Following line creates an instance of the "universal" dialog:
OpenConnDlg conn;

// No the function show() of this dialog is called.
// This causes the display of a connection directory or the
// automatically attaching to an already existing connection.
// Connection establishment, logon, retrieval of Hicom language and
// system variant are done automatically.

// If everything works well this function returns value 1 and "our"
// application can begin.

if (conn.show())
{
    // Now we can send the first command to Hicom.
    // Therefor the function Send of the predefined object named "comwin"
    // is called.
    // return value of this function is the complete AMO result of the
    // passed command.

    // Before we send the command we define a string variable named
```



```
// "result" that should take the AMO result. Values of of string
// variables can be unlimited in size (no limitation to
// 255 characters or similar restrictions)

string result;

// Now we let the Hicom execute "REGEN-SCSU;"
result=comwin.Send("REGEN-SCSU;");
// The result shall now be processed. In an array of strings each
// command of the regeneration result shall be stored.

// The array of strings named "cmds" is declared in this way:

string [] cmds;

// Using the functions sys.prepare_regen_results (from sysfunc.cbl)
// the AMO result gets parsed:
cmds=sys.prepare_regen_results(result);

// Now every command should be processed in a loop.
// Therefor we need a loop variable:

int i;

// With this loop construct we iterate over every command of the
// 'cmds' array

for i=0 to cmds.count-1 do
{

    // The job to parse the AMO command is passed to ComWin-Edit.
    // The predefined object comedit has already got the Hicom Variant
    // and Language by openConnDlg.
    // With this knowledge it is able to parse the AMO syntax.

    comedit.HicomCommand=cmds[i];
    // From comedit we can get every value of the different parameters.
    // And we can send this value e. g. into the ComWin-Connect
    // protocol window.

    comwin.Print(comedit.getParam("TLNNU")
                + " --> "
                + comedit.getParam("LAGE")
                + "\n" );
}
```

Introduction into CBL Programming

HiPath 4000 Expert Access Interoperation

```
// (character strings can be concatenated with + operator.)

} // for

// The same for SBCSU:

result=comwin.Send("REGEN-SBCSU;");
cmds=sys.prepare_regen_results(result);
for i=0 to cmds.count-1 do
{
    comedit.HicomCommand=cmds[i];
    comwin.Print(comedit.getParam("TLNNU")
        + " --> "
        + comedit.getParam("LAGE")
        + "\n" );
}
}
```

Index

Symbole

- 190
-- 13
!= 193
% 190
& 191
&& 189
* 190
*= 13
+ 190
++ 13
+= 13
/ 190
/* 49
// 49
/= 13
-= 13
== 193
> 193
>= 193
^ 191
_BeautifyXML 74
_Busy (ComWin-Connect) 114
_Defined (interpreter functions) 73
_HttpDownloadFile (interpreter functions) 73
_Import 62
_ReleaseWait (interpreter functions) 73
_WaitUntilReleaseWait (interpreter functions) 73
| 191
|| 190

A

Active (Comedit) 97
AfrSvr.exe 144
Alert (ComWin-Connect) 114
align (cblFormControl) 156
AmoParameter control 162
AmoResult (Comedit) 97
AmoResultPos (Comedit) 97

AND 189, 191
ApplicationLanguage (Comedit) 97
Args 71
Arithmetic operators 190
Array type 182
ArrayRemoveAt (interpreter functions) 73
arrays 51
ASN.1 67
assign statement 13
Assignment operators 193
Attach (ComWin-Connect) 114
Authenticate (ComWinAccessSession) 138

B

bitmap 163
Bitwise operators 191
Boolean operators 188
Break (ComWin-Connect) 114
break (keyword) 50
break statement 37
breakFlag 149
BringToFront (ComWin-Connect) 114
button 162

C

caption (cblForm) 149
caption (cblFormControl) 153
catch (keyword) 50
CBL 176
CBL Language 10
CBL Programmer's Reference 71
CBL Programming 176
cbl_chain 89
cbl_dialogs 93
cbl_sysfunc_ 76
cblFile 90
cblForm 148
cblFormControl 152
cblIniFile 92
cblMenu 168
cblsys1.cblIniFile 92
cblsys1.cblZipFile 93
cblsys1.chain 89
cblsys1.dialogs 93
cblsys1.dll 76

- cblsys1.file 90
- cblsys1.sysfunc 76
- cblsys1.xmlfile 94
- cblXmlFile 94
- cblXmlRecord 94
- cblZipFile 93
- chain_to_string (cbl_chain) 89
- char (sys) 76
- Check (Comedit) 97
- check listbox 161
- checkbox 162
- CheckComplete (cblForm) 150
- CheckControl (cblForm) 151
- CheckFileUntilError (Comedit) 98
- CheckParam (Comedit) 97
- class (keyword) 50
- class hierarchy 27
- class statement 25
- clock (sys) 77
- close (cblFile) 90
- Close (cblForm) 151
- Close (cblIniFile) 92
- close (cblXmlFile) 95
- close (cblZipFile) 93
- Close (ComWin-Connect) 114
- CloseAfterAutomation (ComWin-FT) 131
- CloseAll (ComWin-Connect) 114
- CloseDatabase (ComWinAccessDatabase) 140
- col (cblFormControl) 154
- colspan (cblFormControl) 155
- combo edit field 160
- Comedit 96
- comedit (predefined object) 98
- Comedit.MMLCommand 96
- command (Comedit) 97
- Command line parameters 71
- comment (single line) 49
- comment (structured) 49
- comwin (predefined object) 115
- ComWin.Control 114
- ComWin.FileTransfer 131
- ComWin.FM 144
- ComWin.Frame 125

- ComWin.PPPMan 136
- ComWin300 Architecture 7
- ComWinAccessDatabase 140
- ComWinAccessFileman 139
- ComWinAccessQuery 141
- ComWinAccessServer 138
- ComWinAccessSession 138
- ComwinControl 114
- ComWinDbAccess.Database 140
- ComWinDbAccess.Fileman 140
- ComWinDbAccess.Query 141
- ComWinDbAccess.Server 138
- ComWinDbAccess.Session 138
- ComWinFM 144
- ComwinFrame 125
- comwinFrame (predefined object) 125
- ComWinFT 130
- ComWinPPP 136
- Connect (ComWin-FT) 131
- Connected (ComWin-FT) 131
- Connected (ComWin-PPP) 136
- Connections (ComWin-Connect) 114
- ConsoleClear (ComWin-Frame) 125
- ConsoleHide (ComWin-Frame) 125
- ConsolePrint (ComWin-Frame) 125
- ConsolePrintStyle (ComWin-Frame) 125
- ConsoleProgress (ComWin-Frame) 125
- ConsoleResize (ComWin-Frame) 125
- ConsoleShow (ComWin-Frame) 125
- const (keyword) 50
- ContainsStrIndex (interpreter functions) 72
- controls 152
- controls (cblForm) 149
- CopyFrom (Comedit) 98
- count (keyword) 50
- create (cbl_chain) 89
- CreateSessionToken (ComWinAccessSession) 138
- CreateTempDir (sys) 77
- CreateTempFile (sys) 77
- ctAmoParameter 162
- ctBitmap 163
- ctButton 162
- ctCheckbox 162

ctCheckListBox 161
ctCombo 160
ctEdit 159
ctLabel 159
ctListBox 160
ctMemo 160
ctPages 163
ctPanel 163
ctRadiogroup 161
currentDateTime (sys) 77

D

data types 179
Database (ComWinAccessQuery) 141
declaration statement 22
decrypt (sys) 77
Delete (ComWin-FT) 131
DeleteTempDir (sys) 77
DestinationProcessor (Comedit) 97
DestType (ComWin-PPP) 136
Device (ComWin-PPP) 136
DialogMode (ComWin-FT) 131
DialogMode (ComWin-PPP) 136
Directory (cbl_dialogs) 94
directory (cblZipFile) 93
Disconnect (ComWin-FT) 131
Dispose (interpreter functions) 73
DLL interface 60
dll_class (keyword) 50
do (keyword) 50
do_while statement 33

E

edit field 159
else (keyword) 50
elseif (keyword) 50
empty statement 17
enabled (cblForm) 149
enabled (cblFormControl) 153
encrypt (sys) 77
enum (keyword) 50
enum statement 15
eof (cblFile) 90
Equality operators 193
error (cblFormControl) 154

Error (Comedit) 97
ErrorLine (Comedit) 98
ErrorMsg (Comedit) 97
ErrorParam (Comedit) 97
ErrorParamIdx (Comedit) 97
ErrorValue (Comedit) 97
ExecSql (ComWinAccessQuery) 141
ExistsParam (Comedit) 97
Expression 197
expression 39
expression lists 44
extends (keyword) 50
ExtractFilePath (sys) 77

F

FieldByIndex (Comedit) 98
FieldByName (Comedit) 98
FieldCount (Comedit) 98
FieldName (Comedit) 98
file_compare (sys) 77
FileDate (sys) 78
FileName (cbl_dialogs) 93
FilePos (Comedit) 98
FileSplit (ComWin-FT) 131
FileUnsplit (ComWin-FT) 131
Filter (cbl_dialogs) 94
FindFile (sys) 77
FindMask (Comedit) 97
flow control 197
FocusControl (cblForm) 151
for - statement 200
for (keyword) 50
for statement 31
ForceDirectories (sys) 77
form.cbl 146
format (sys) 77
Function - statement 198
function call 14
function declaration 19
function parameters 20

G

GarbageCollection (interpreter functions) 72
Get (cblForm) 151
Get (ComWin-FT) 131

GetControl (cblForm) 151
 GetDir (ComWin-FT) 131
 GetErrorMsg (ComWinAccessDatabase) 140
 GetErrorMsg (ComWinAccessQuery) 141
 GetFamosUserPass (ComWinAccessSession) 138
 GetFile (ComWinAccessFileman) 140
 GetFirst (ComWinAccessQuery) 141
 GetHicomVariants (Comedit) 97
 GetHosts (ComWin-FT) 131
 GetInfo (ComWinAccessSession) 138
 GetInfo (ComWin-FT) 131
 GetNext (ComWinAccessQuery) 141
 GetNextMask (Comedit) 97
 getParam (Comedit) 96
 getParamDesc (Comedit) 97
 GetParameters (ComWin-Connect) 114
 getParamIndex (Comedit) 97
 getParamName (Comedit) 96
 getParamNames (Comedit) 97
 getParamProps (Comedit) 97
 getParamValidation 162
 getParamValidation (Comedit) 97
 getValue (Comedit) 97
 getValueDesc (Comedit) 97
 GUI-Builder 209
 GuiBuilder 8

H

height (cblFormControl) 155
 helpContext (cblForm) 149
 helpContext (cblFormControl) 158
 hex2int (sys) 76
 Hicom/HiPath error messages 144
 HicomCommand (Comedit) 97
 HicomLanguage (Comedit) 97
 HicomLanguage (ComWin-Connect) 115
 HicomVariant (Comedit) 97
 HicomVariant (ComWin-Connect) 115
 Hide (ComWin-Connect) 114
 hint (cblFormControl) 153
 HostDesc (ComWin-Connect) 115
 HostDispName (ComWin-Connect) 115

HostName (ComWin-Connect) 115
 Hosts (ComWin-Connect) 114
 HTSOpen (ComWin-Connect) 114

I

icon (cblForm) 149
 Id (ComWinAccessDatabase) 140
 Id (ComWinAccessServer) 138
 Id (ComWinAccessSession) 138
 Id (ComWin-PPP) 136
 Identifier 185
 identifier 46
 if - statement 199
 if (keyword) 50
 if statement 29
 import (keyword) 50
 import statement 18
 in (keyword) 50
 include (keyword) 50
 include statement 17
 index (keyword) 50
 Init (ComWin-FM) 144
 int (keyword) 50
 int2hex (sys) 76
 Integer type 179
 isMainForm (cblForm) 149
 items (cblFormControl) 158

K

KeywordOriented (Comedit) 97
 keywords 50

L

label text 159
 LastMessage (ComWin-Connect) 114
 Layout 146
 left (cblFormControl) 155
 lfill (sys) 76
 listbox 160
 LocalFile (ComWin-FT) 131
 Logoff (ComWin-Connect) 114
 Logon (ComWin-Connect) 114
 ltrim (sys) 76

M

- MacroRunning (ComWin-Connect) 114
- mainmenu (cblForm) 149
- match_regex (sys) 76
- memo control 160
- menu 168
- Message (ComWin-Connect) 114
- MessageBeep (sys) 77
- MessageDialog (cbl_dialogs) 94
- Modal dialog example 173
- modified (cblFormControl) 154

N

- namespace 19
- namespace (keyword) 50
- National Language Support 174
- NewLine (ComWin-Connect) 114
- NLS 174
- Noun (Comedit) 97
- null (keyword) 50

O

- Object and Class 203
- oldValue (cblFormControl) 154
- ole_class (keyword) 50
- onChange (cblFormControl) 158
- onClick (cblFormControl) 158
- onClose (cblForm) 149
- OnDataAvailable (ComWin-FM) 144
- onEnter (cblForm) 149
- onHelp (cblForm) 149
- onHelp (cblFormControl) 158
- onNavigate (cblFormControl) 158
- open (cblFile) 90
- Open (cblIniFile) 92
- open (cblXmlFile) 95
- open (cblZipFile) 93
- Open (ComWinAccessServer) 138
- Open (ComWin-Connect) 114
- OpenDatabase (ComWinAccessDatabase) 140
- OpenDialog (cbl_dialogs) 93
- OpenDirect (ComWin-Connect) 114
- Operator - 42

- Operator + 41
- operator ++ 13
- operator += 13
- Operator = 43
- operator -= 13
- Operator in 42
- Operators 188
- operators *= 13
- OR 190, 191
- outofdate (cblFormControl) 154

P

- panel 163
- panels (cblFormControl) 158
- param_intervals (sys) 77
- param_ivals (sys) 77
- ParamCount (Comedit) 97
- parse_amo_errors (sys) 77
- parse_amo_result (sys) 77
- ParseAmoResult (Comedit) 98
- ParseUrl (sys) 78
- Password (ComWin-PPP) 136
- PhoneNumber (ComWin-PPP) 136
- popupMenu (cblFormControl) 159
- PPP-Manager 136
- precedence 196
- prepare_regen_results (sys) 77
- Print (ComWin-Connect) 114
- PrintLog (ComWinAccessSession) 138
- Profile (ComWin-PPP) 136
- program 10
- Put (ComWin-FT) 131

R

- radio buttons 161
- radio group 161
- RasConnError (ComWin-PPP) 136
- RasConnState (ComWin-PPP) 136
- RasStatusString (ComWin-PPP) 136
- read (cblFile) 90
- read (cblXmlFile) 95
- ReadInteger (cblIniFile) 92
- readln (cblFile) 90
- readonly (cblFormControl) 153
- ReadString (cblIniFile) 92

- reference variables 53
- RegistryExistsKey (sys) 77
- RegistryGetKeyNames (sys) 77
- RegistryReadInteger (sys) 77
- RegistryReadString (sys) 77
- RegistryWriteInteger (sys) 77
- RegistryWriteString (sys) 77
- RemoteFile (ComWin-FT) 131
- remove_chain_from_chain (cbl_chain) 89
- remove_int_from_chain (cbl_chain) 89
- Rename (ComWin-FT) 131
- replace (sys) 76
- required (cblFormControl) 153
- ResetModified (cblForm) 151
- ResetModifiedControl (cblForm) 151
- RetrieveSizePos (cblForm) 152
- return (keyword) 50
- return statement 36
- row (cblFormControl) 154
- rowspan (cblFormControl) 155
- rtrim (sys) 76

S

- SaveDialog (cbl_dialogs) 93
- ScanMask (Comedit) 97
- Scope of variables 187
- Script (ComWin-PPP) 136
- Send (ComWin-Connect) 114
- SendEx (ComWin-Connect) 114
- Server (ComWinAccessDatabase) 140
- Server (ComWinAccessFileman) 140
- Server (ComWinAccessQuery) 141
- Server (ComWinAccessSession) 138
- ServerIp (ComWin-PPP) 136
- Session (ComWinAccessDatabase) 140
- Session (ComWinAccessFileman) 140
- set (keyword) 50
- set statement 16
- SetFileDate (sys) 78
- setParam (Comedit) 96
- SetParam (ComWinAccessQuery) 141
- SetProtocol (ComWin-FT) 131
- SetStatus (cblForm) 151
- SetWindowState (cblForm) 152
- ShellExecute (sys) 77
- Show (cblForm) 150
- Show (ComWin-Connect) 114
- ShowBrowser (ComWin-FT) 131
- ShowCommand (Comedit) 97
- ShowCommandDlg (Comedit) 97
- ShowHelp (ComWin-Frame) 125
- ShowModal (cblForm) 150
- showModified (cblForm) 149
- showModified (cblFormControl) 154
- ShowParamDlg (Comedit) 97
- ShowProgress (interpreter functions) 73
- signature statement 17
- SkipLines (Comedit) 98
- SkipLinesUntil (Comedit) 97
- sort_grid (sys) 76
- speedbar (cblForm) 149
- StartApplication (ComWin-Frame) 125
- StartJavaServer (interpreter functions) 72
- state (cblForm) 149
- statement 11
- Statement flow control 197
- status (cblForm) 149
- Status (ComWin-Connect) 114
- Status (ComWin-PPP) 136
- StopJavaServer (interpreter functions) 72
- StoreSizePos (cblForm) 152
- strfind (sys) 76
- string (keyword) 51
- String type 181
- string_lines (sys) 76
- string_token (sys) 76
- stripquotes (sys) 76
- strlen (sys) 76
- substr (sys) 76
- super (keyword) 51
- supportUndo (cblFormControl) 154
- syn (keyword) 51
- syn (predefined object) 78
- syn statement 14
- Syntax description 10
- syslib (predefined object) 78
- system (sys) 77

T

- taborder (cblFormControl) 158
- tag (cblFormControl) 158
- text tables 174
- text_table (keyword) 51
- text_table statement 16
- TextHeight (interpreter functions) 72
- TextWidth (interpreter functions) 72
- this (keyword) 51
- throw (keyword) 51
- throw statement 34
- to (keyword) 51
- tolower (sys) 76
- top (cblFormControl) 155
- toupper (sys) 76
- TracePrint (interpreter functions) 73
- translate (cblFormControl) 154
- Translateln (interpreter functions) 72
- TranslateOut (interpreter functions) 72
- trim (sys) 76
- try (keyword) 51
- try_catch statement 34
- type (cblFormControl) 153
- type identifier 185
- type-identifier 45

U

- unzip (cblZipFile) 93
- Update (cblForm) 150
- UpdateControl (cblForm) 150
- User (ComWin-PPP) 136

V

- value (cblFormControl) 158
- Variables 185
- variant (keyword) 51
- Verb (Comedit) 97
- virtual (keyword) 51
- visible (cblForm) 149
- visible (cblFormControl) 154
- Visible (ComWin-FM) 144
- Visible (ComWin-FT) 131
- Visible(ComWin-PPP) 136
- void (keyword) 51

W

- Wait (interpreter functions) 72
- WaitForMessages (ComWin-FM) 144
- while - statement 202
- while (keyword) 51
- while statement 32
- width (cblFormControl) 155
- WindowState (ComWin-Connect) 114
- with (keyword) 51
- with statement 35
- write (cblFile) 90
- write (cblXmlFile) 95
- WriteInteger (cblIniFile) 92
- WriteString (cblIniFile) 92

X

- XOR 191
- XSD 64

