Oberon as an Implementation Language for COM Objects

Jurg Gutknecht, ETH Zurich
E-mail address: gutknecht@inf.ethz.ch

Abstract

This is a short report on a short project, carried out during a 3-month sabbatical stay at Microsoft Research in the fall of 1998. The motivation for this endeavor was doublefold: (a) verify the degree of language independence of the COM component technology and (b) explore COM as a potential commercial environment and framework for Oberon applications. The project finally converged towards a simple case study, an electronic accounting system implemented in Oberon as a COM server under Windows NT and illustratively used by a Visual Basic client.

Keywords: Object, Component, COM, Oberon

Oberon as an Isolated System

Project Oberon [1] was started in 1986 by Niklaus Wirth and the author of this report. Oberon as a language [2] extends the line of Pascal and Modula-2, Oberon as a system is a fully modular and compact operating system for personal computers. The first Oberon implementation ran on Ceres, a proprietary National Semiconductor 32000 based hardware platform. From this original version, several evolutions and numerous implementation variants have emerged. For example, one native implementation (for Intel PCs) and four embedded implementations (for Windows, Macintosh, Linux, and Unix on Intel, PowerPC and SPARC hardware) of Oberon System 3 [3] exist today.

Even though Oberon is an attractive software development language and system, it is hardly used in commercial projects because of its incompatibility with the standards. In essence, neither the Oberon system nor products developed with it are able to communicate with any non-Oberon host environment in a nontrivial way. It is therefore desirable to adapt Oberon to a universal component framework like COM (Common Object Model) [4, 5] that enables components (sometimes called *controls*) from different cultures to contribute in an organized and orderly way to some coherent, global functionality.

Oberon as a COM Participant

In order to explain the essential steps towards Oberon as a COM participant, we first recall that COM uses an elaborate interface concept in order to decouple component implementations from clients. Its two main aspects correspond directly to the two main aspects of adapting Oberon (or any other language) to COM. They are (a) the IDL (Interface Description Language) formalism used to specify individual interfaces and (b) the "1 object implements n (immutable) interfaces" relation.

IDL is a notation that is used in slightly different variations in high-level object-oriented frameworks like CORBA and COM to describe interfaces formally but independent of the implementation language [6]. However, in practice, IDL explicitly or (worse) implicitly dictates a variety of nuances and details "behind the scenes" like memory mapping of data types and procedure calling conventions that implementation languages have to comply with. As IDL is close in style and spirit to the C/C++/Java language family, members of this family are in favor compared to, for example, Oberon. Concretely, the following problems occurred in the course of our project: (a) the need for data types not supported by Oberon (for example, unsigned integers and bit arrays of different sizes), (b) the need for tag-free structured types (structured types in Oberon carry a tag to support run-time typing and garbage collection), and (c) the need for procedure calling protocol incompatible with Oberon (for example, parameters to be popped from stack by caller). We solved these problems ad hoc by (a) mapping unsupported IDL data types to closest Oberon data types, (b) introducing a "no tag" compiler directive for record types and array types, and (c) introducing a "standard call" compiler directive for procedures and functions.

Examples

```
TYPE
  UINT = LONGINT;
  WORD = INTEGER;
  DWORD = LONGINT;
  DISPID = LONGINT;
```

```
  ArgList = RECORD [notag]
    arg: ARRAY MaxArgs OF VARIANT
  END;

  DISPIDList = RECORD [notag]
    id: ARRAY MaxArgs OF DISPID
  END;

  DISPPARAMS = RECORD [notag]
    rgvarg: POINTER TO ArgList;
    rgdispidNamedArgs: POINTER TO DISPIDList;
    cArgs, cnamedArgs: UINT
  END;

PROCEDURE QueryInterface [stdcall] (this: Interface; iid: GUID;
  VAR ifc: Interface): HRESULT;
```

The second major aspect to be dealt with is the “1 object implements n interfaces” relation, where every implemented interface corresponds to a certain “facette” of the object. In the cases of C++ and Java this relation is modeled as (multiple) definition inheritance. In the case of Oberon, we have chosen an approach based on a more explicit management of interfaces by objects. Interfaces in COM Oberon are instances of the following type:

```
TYPE
  Interface = POINTER TO RECORD
    vtbl: IUnknownVT;
    obj: Object;
    next: Interface;
    iid: GUID
  END;
```

vtbl is a pointer to a v-table corresponding to the IUnknown interface or any extension thereof, *obj* refers to the underlying object implementing this interface, *next* serves as a link connecting the set of interfaces implemented by the underlying object, and *iid* is the GUID of this interface.

v-tables in COM Oberon are record structures consisting of function pointers. The Oberon type extension construct is used to denote interface extensions.

Example

```
TYPE
  IUnknownVT = POINTER TO RECORD
    QueryInterface: PROCEDURE [stdcall] (this: Interface; iid: GUID;
      VAR ifc: Interface): HRESULT;
    AddRef: PROCEDURE [stdcall] (this: Interface): LONGINT;
    Release: PROCEDURE [stdcall] (this: Interface): LONGINT
  END;

  IPersistVT = POINTER TO RECORD (IUnknownVT)
    GetClassID: PROCEDURE [stdcall] (this: Interface;
      VAR clsid: GUID): HRESULT
  END;
```

COM Oberon objects are again instances of a pointer based structured type:

```
TYPE
  Object = POINTER TO RECORD
    prev, next: Object;
    mcref: LONGINT;
    ifc: Interface
  END;
```

mcref subsumes the COM reference count. The entirety of objects currently in use by COM (i.e. with mcref > 0) are linked to a ring via *prev/next* in order to prevent the Oberon garbage collector from reclaiming objects used by COM but not by Oberon. Finally, the *ifc* in field in Object refers to the list of interfaces implemented by this object.

Remarks

(1.) The connections of implementations with v-table entries and interfaces with objects are explicitly made at object creation time.

(2.) The IUnknown interface is an implicit part of every member in the list of interfaces supported by an object. Additionally, IUnknown is included in this list as an explicit member, in order to guarantee unique identifyability of the object by querying it for IUnknown.

(3.) The reference counting mechanism could easily be refined, i.e. the reference count that is now centrally stored in an object could easily be distributed to the different interfaces that it supports.

(4.) Unlike definition inheritance models, our run-time model allows objects to implement multiple instances of the same interface without the help of additional (complex) constructions like nested classes. This is useful, for example, in the case of multiple connection point interfaces provided by a server object. As a fine point note that connection point interfaces are not included in the list of interfaces implemented by an object, because they are retrieved directly via the FindConnectionPoint method rather than via their GUID.

One interesting point about COM is its universality not only in terms of language independence but also in terms of operating system independence. Nevertheless, Windows is the natural platform for COM, so that we decided to use Windows (NT, to be more precise) as target platform for COM Oberon, both as a development system and a runtime system. Consequently, we had to provide (a) an Oberon definition layer connecting Oberon with the basic Windows API and (b) a tool to generate a Windows compatible run-time format from Oberon modules.

The Oberon definition of the Windows API consists of two modules Kernel32 and ADVAPI. They are used by low-level Oberon modules, notably by Kernel, Registry and Console. Module Kernel is Oberon's own kernel implementing garbage collection, object finalization etc., module Registry supports registration of (keyword, value) pairs in the Windows registry, and module Console allows simple trace output from Oberon to the Windows console. The run-time format generator in its current form is called PELinker. It simply links (compiled) Oberon module hierarchies to portable Windows executables (i.e. DLL and exe-files in PE format).

Case Study: An Electronic Accounting System as a COM Oberon control

The choice of an application, i.e. an exemplary implementation of a COM control in Oberon was guided by the following list of checkpoints:

(1.) The application code should be compact. The COM-related code should dominate the implementation.

(2.) The application code should not be trivial. It should include dynamic data structures beyond the list type and fast algorithms that are not typically available in higher-level client languages like Visual Basic.

(3.) The control should produce special events resulting from its history rather than from some individual call or use.

We finally decided for an electronic accounting system that is able to efficiently manage a potentially huge number of accounts and that provides the following functionality:

(1.) Open (new) account with given id (name), initial balance (positive or negative) and lower limit (negative).

(2.) Query balance of specified account.

(3.) Add/withdraw specified amount to/from specified account.

With that, our checkpoints were easily met: We simply implemented the accounting system as an ordered binary tree and defined bankrupcy as a special event.

The multicultural character of COM and its benefits can perhaps be demonstrated to its best, if the client culture is "higher" than the implementation culture of the server (Oberon in our case). From the perspective of a high-level client, any lower-level server appears as an "own small world" that does not have to be understood in detail. The server simply serves a certain purpose, defined by its interface(s).

In our case study, we chose Visual Basic as client environment and container for our COM Oberon accounting control. The following Visual Basic project is a possible implementation of a client:

Form1

```
Private Sub Command1_Click()

  Dim x As Object
  Dim e As New Class1
  Dim f As New Class2

  Set x = CreateObject("COMOberon")
  MsgBox x
  x = "My E-Bank"
  MsgBox x
  MsgBox "OWN " & x.Open("OWN")
  MsgBox "X " & x.Open("X", 500, -1000)
  MsgBox "Y " & x.Open("Y", 2000, -2000)
  MsgBox "X " & x.Trans("X", 200)
  MsgBox "Y " & x.Trans("Y", 100)
  MsgBox "X " & x.Trans("X", 400)
  MsgBox "Y " & x.Query("Y")
  MsgBox "Y " & x.Trans("Y", -3300, f)
  MsgBox "Y " & x.Trans("Y", -3300, e)
  MsgBox "A " & x.Open("A", 10000, -1000)
  MsgBox "B " & x.Open("B", -800, -2000, e)
  MsgBox "X " & x.Trans("X", -2500, e)
  MsgBox "C " & x.Trans("C", 800)
End Sub
```

Class1

```
Implements IEventHandling

Dim reserve As Integer

Public Sub IEventHandling_HandleBankrupcy(x As Variant, ByVal ub As Long)
  If reserve \ 100 >= -ub Then amt = -ub Else amt = reserve \ 100
  MsgBox x.Trans("OWN", amt) & " added"
  reserve = reserve - amt
End Sub

Private Sub Class_Initialize()
  reserve = 8000
End Sub
```

Class2

```
Public Sub HandleBankrupcy()
  MsgBox "Bankrupt"
End Sub
```

Apart from some standard COM interfaces like IUnknown and IPersistPropertyBag, Visual Basic communicates with its servers via the IDispatch interface. IDispatch is a generic interface that is based on run-time dispatching of property accesses, event reports, and general method calls, including parameter checking and decoding. The IDispatch interface provided by our accounting control is implemented by InvokeImpl (see source listing in the Appendix). It provides one (persistent) property (its name) and three methods (Open, Query, and Trans(action)). The parameter lists of the methods are dynamic in the sense that their tail-ends can be omitted. All parameters are of the standard type VARIANT. The variants currently supported by COM Oberon are VARIANT (again), BSTR (Basic string), integers, long integers, and IDispatch interfaces.

The IDispatch parameter variant is used by our accounting control to pass an *event agent* that is supposed to handle the bankrupcy event. It is noteworthy that such objects are passed by the client to the server, i.e. that they are typically implemented in the client's language. In fact, delegating a task to possibly multiple servers while still keeping control over the handling of special events is a highly successful software design pattern propagated by Visual Basic and emphasizing its role as "glue logic".

For the purpose of illustration, we have implemented two different kinds of event agents corresponding to two different Visual Basic classes and to two different calling techniques. The essential point in both cases is the fact that the COM Oberon object now acts as a client calling a “source” interface (or “outgoing” interface) that is implemented elsewhere, in this case on the side of the event agent.

Our COM Oberon control defines its source interface explicitly and formally by means of a type library and by corresponding entries in the Windows registry. The following *odl* description (to be compiled into the type library by the *midl* compiler) reveals that it is an IDispatch interface called IEventHandling:

```
[uuid (51238A13-75D2-11D2-9303-00A0C99334B2),
  helpstring("COM Oberon type library"), lcid(0), version(1.0)]

library COMOberon {
  importlib("STDOLE32.TLB");

  [uuid (51238A14-75D2-11D2-9303-00A0C99334B2), version(1.0),
oleautomation]
  dispinterface IEventHandling {
    properties:
    methods: [id(1)] void HandleBankrupcy (VARIANT* x, int ub);
  };

  [uuid(8C74F345-6A54-11D2-9302-00A0C99334B2),
    helpstring("COM Oberon control"), version(1.0)]

  coclass COMOberon {
    [default, source] dispinterface IEventHandling;
 };
};
```

IEventHandling features just one method HandleBankrupcy that takes one variant parameter and one integer parameter. The variant parameter refers to the IDispatch interface of the COM Oberon control that generated the event and thereby enables the event agent to communicate back.

Note now that Class1 (see the Visual Basic client code above) implements the IEventHandling interface, thereby making beneficial use of the callback possibility by transferring the lacking amount to the OWN account of the event-causing control (up to a certain percentage of the global reserves). Class1 event agents (like e) therefore guarantee the availability of the outgoing interface, so that our COM Oberon control can proceed as planned.

The situation is different for event agents like f of a class that does not (explicitly) implement IEventHandling. In this case, the COM control needs to find a different way of calling the agent. An obvious solution is to use the IDispatch interface that is generated implicitly by the Visual Basic compiler and that, by definition, comprises the set of public subroutines of the class. Note that in this case the calling COM Oberon control has to carry the full burden of checking consistency between call and called implementation. This in contrast to the previous case, where the check is done by the Visual Basic compiler using the type library.

The strategy used by the COM Oberon control is this: If the class of the event handling object implements the interface defined by the type library, use it, otherwise assume that a simple, parameterless variant of HandleBankrupcy is implemented. In terms of run-time COM, this strategy is expressed as follows (for details see procedure Bankrupt in the source listing in the Appendix):

```
{ Get event agent’s IDispatch interface I from parameter;
  Query I for interface IEventHandling as defined by the type library;
  If answer is positive, use IEventHandling
  Else try to get dispid of HandleBankrupcy from I;
    If answer is positive, use I
    Else apply default event handling
}
```

Remarks

(1.) The inclusion of an explicit event agent in the interface between client and server is not an optimal solution. If our COM Oberon control would qualify as an ActiveX control (i.e. if it would support the necessary

interfaces), the Visual Basic container would by its own establish an event connection with our control via the IConnectionPointContainer mechanism. Note that our implementation is prepared for this.

(2.) Our accounting system control uses the IPersistPropertyBag mechanism to save its only property (its name) persistently. However, no provision is made yet for the actual contents of our control (i.e. the collection of accounts) to be persistent. This would be easy with the direct use of Oberon's file system. However, we aim at an implementation that consequently uses IPropertyBag.

(3.) Our COM Oberon control is currently embedded into a DLL, and it therefore appears as an "in process" server. Thanks to COM's transparency along the process distance axis, we do not anticipate problems with porting it to an "out of process" server, but this would be a worthwhile experiment.

Conclusion and Outlook

We have designed and implemented a prototype of a fully grown COM control in Oberon that is ready to be used by any COM client and in particular by Visual Basic and HTML. The implementation is based on an underlying compiler/linker/run-time framework connecting Oberon with Windows. Our experience has shown that COM is in fact an appropriate technology to serve as a sound basis for a seamless integration of a "foreign" language like Oberon (not necessarily belonging to the C-"family") into a widespread operating system like Windows or, in other words, to serve as a viable environment for "small worlds" implemented in a foreign language.

However, we consider our COM Oberon control a prototype mainly because Oberon is still lacking any institutionalized framework that supports the COM interface mechanism. For example, Oberon's arsenal of types is not rich enough to provide a full image of the IDL data types. Also, the compiler directive mechanism allowing standard calls and tag-free structures is tentative. What is actually needed is comprehensive support by both the language and the compiler for (a) interfaces specified in IDL and (b) the "object implements interface" relation. We plan to design and develop such support in the near future. However, we do not plan to map the "implements" relation to multiple interface inheritance along the C++/Java lines.

We further plan to combine the development just mentioned with our "active object" concept. Active objects are self-controlled (in contrast with remote-controlled) by one or multiple in-scope light-weight processes and are therefore fully self-contained. We believe that the "active object" approach is an interesting alternative (or enhancement) to the currently dominating "inverse programming" approach, presenting itself in the clothes of event handling.

Acknowledgement

My foremost thanks go to Tony Williams for providing a most interesting and effective but no less enjoyable environment for my 3-month stay at Microsoft Research. I am very impressed of the high standards set and maintained by Tony and his team. My warmest thanks also go to the other members of Tony's group and in particular to Crispin Goswell and Dennis Canady for their valuable and competent help with COM and Visual Basic, without it I would not have been able to successfully complete my prototype. I also won't forget many inspiring discussions with Tony, Walt Hill and Charles Frankston.

Last but certainly not least, I gratefully acknowledge the truly "fundamental" contribution of my collaborator Emil Zeller. His "Windows Oberon" implementation now includes the compiler adjustments, the linker and the API-kernel mentioned in the text. Thanks to his profound knowledge of both the Windows culture and the Oberon culture and to his super-efficient way of working, remote control from Redmond to Zurich worked well in this case.

References

[1] Wirth, N., Gutknecht, J., *Project Oberon - The Design of an Operating System and Compiler* , Addison-Wesley, 1992, ISBN 0-201-54428-8.

[2] Reiser, M., Wirth, N**.,** *Programming in Oberon - Steps Beyond Pascal and Modula*, Addison-Wesley, 1992, ISBN 0-201-56543-9.

[3] Fischer, A., Marais, H., *The Oberon Companion: A Guide to Using and Programming Oberon System 3*, vdf Hochschulverlag AG, ETH Zurich, 1998, ISBN 3-7281-2493-1.

[4] C. Szyperski, ***Component Software*** - *Beyond Object-Oriented Programming***,** Addison-Wesley, 1998, ISBN 0-201-17888-5.

[5] Box, D., *Essential COM*, Addison-Wesley, 1998, ISBN 0-201-63446-5.

[6] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 1.1*., 1992.

Appendix

Full Source Code of the COM Oberon Implementation

```
MODULE COMOberon; (*Electronic Accounting System/ JG 11/25/98*)
  IMPORT SYSTEM, Kernel, Kernel32, Registry, Modules, Console;

  CONST
    (*HRESULT codes*)
    SOk = 0H; SFalse = 1H;
    ENoInterface = 080004002H;
    EFail = 080004005H;
    ClassENoAggregation = 080040110H;
    ClassEClassNotAvailable = 080040111H;

    (*IDispatch*)
    (*DISPATCHMETHOD: BIT 0;
    DISPATCHPROPERTYGET: BIT 1;
    DISPATCHPROPERTYPUT: BIT 2;*)
    VTBSTR = 8; (*variant = BASIC string*)
    VTI4 = 3; (*variant = 32 bit int*)
    VTI2 = 2; (*variant = 16 bit int*)
    VTDISPATCH = 16393; (*16384 + 9*)
    VTVARIANT = 16396; (*16384 + 12*)

    (*registry data*)
    CLSIDCOMOberon = "{8C74F345-6A54-11D2-9302-00A0C99334B2}";
    TLBIDCOMOberon = "{51238A13-75D2-11D2-9303-00A0C99334B2}";
    OIIDCOMOberon = "{51238A14-75D2-11D2-9303-00A0C99334B2}";
    TLBPathName = "C:\Program Files\DevStudio\Vc\bin\COMOberon.tlb";
    ObjectCaption = "COMOberon Object";
    InterfaceCaption = "COMOberon EventHandling";
    ProgID = "COMOberon.1";
    VIndProgID = "COMOberon";
    BitmapID = "COMOberon, 1";
    VersionNo = "1.0";

    MaxLen = 50; MaxFuncs = 5; MaxArgs = 5;

  TYPE
    UINT = LONGINT;
    WORD = INTEGER;
    DWORD = LONGINT;
    HRESULT = LONGINT;
    LCID = LONGINT;
    DISPID = LONGINT;
    VARTYPE = INTEGER;
    REFANY = POINTER TO RECORD [notag] END;
    HANDLE = POINTER TO RECORD [notag] ref: REFANY END;
    TYPEINFO = REFANY;
    EXCEPINFO = REFANY;

    GUID = RECORD [notag]
      p1: LONGINT;
      p2, p3: INTEGER;
      p4: ARRAY 8 OF CHAR
    END;

    VARIANTRef = POINTER TO VARIANT;
    VARIANT = RECORD [notag]
      vt: VARTYPE;
```

```
  w1, w2, w3: WORD;
  v1, v2: LONGINT
END;

ArgList = RECORD [notag]
  arg: ARRAY MaxArgs OF VARIANT
END;

DISPIDList = RECORD [notag]
  id: ARRAY MaxArgs OF DISPID
END;

DISPPARAMS = RECORD [notag]
  rgvarg: POINTER TO ArgList;
  rgdispidNamedArgs: POINTER TO DISPIDList;
  cArgs, cnamedArgs: UINT
END;

BSTR = POINTER TO BSTRDesc;
BSTRDesc = RECORD [notag]
  len: LONGINT;
  str: ARRAY MaxLen OF INTEGER
END;

String = ARRAY MaxLen OF CHAR;

(*interface declarations*)

Interface = POINTER TO InterfaceDesc;

IUnknownVT = POINTER TO RECORD
  QueryInterface: PROCEDURE [stdcall] (this: Interface; iid: GUID;
    VAR ifc: Interface): HRESULT;
  AddRef: PROCEDURE [stdcall] (this: Interface): LONGINT;
  Release: PROCEDURE [stdcall] (this: Interface): LONGINT
END;

IDispatchVT = POINTER TO RECORD (IUnknownVT)
  GetTypeInfoCount: PROCEDURE [stdcall] (this: Interface;
    VAR tic: UINT): HRESULT;
  GetTypeInfo: PROCEDURE [stdcall] (this: Interface; tic: UINT;
    lcid: LCID; VAR tinfo: TYPEINFO): HRESULT;
  GetIDsOfNames: PROCEDURE [stdcall] (this: Interface; iid: GUID;
    VAR name: LONGINT; cn: UINT; lcid: LCID;
    VAR dispid: DISPID): HRESULT;
  Invoke: PROCEDURE [stdcall] (this: Interface; dispid: DISPID;
    iid: GUID; lcid: LCID; wFlags: WORD; VAR par: DISPPARAMS;
    VAR res: VARIANT; excpInfo: EXCEPINFO;
    VAR pArgErr: UINT): HRESULT
END;

IConnectionPointContainerVT = POINTER TO RECORD (IUnknownVT)
  EnumConnectionPoints: PROCEDURE [stdcall] (this: Interface;
  VAR ifc: Interface): HRESULT;
  FindConnectionPoint: PROCEDURE [stdcall] (this: Interface;
  iid: GUID; VAR ifc: Interface): HRESULT
END;

IConnectionPointVT = POINTER TO RECORD (IUnknownVT)
  GetConnectionInterface: PROCEDURE [stdcall] (this: Interface;
    VAR iid: GUID): HRESULT;
  GetConnectionPointContainer: PROCEDURE [stdcall] (this: Interface;
    VAR ifc: Interface): HRESULT;
```

```
  Advise: PROCEDURE [stdcall] (this: Interface; ifc: Interface;
    VAR key: DWORD): HRESULT;
  Unadvise: PROCEDURE [stdcall] (this: Interface; key: DWORD): HRESULT;
  EnumConnections: PROCEDURE [stdcall] (this: Interface;
    VAR ifc: Interface): HRESULT
END;

IPersistVT = POINTER TO RECORD (IUnknownVT)
  GetClassID: PROCEDURE [stdcall] (this: Interface;
    VAR clsid: GUID): HRESULT
END;

IPersistPropertyBagVT = POINTER TO RECORD (IPersistVT)
  InitNew: PROCEDURE [stdcall] (this: Interface): HRESULT;
  Load: PROCEDURE [stdcall] (this, bagIfc,
    errorIfc: Interface): HRESULT;
  Save: PROCEDURE [stdcall] (this, bagIfc: Interface;
    clearDirty, saveAll: BOOLEAN): HRESULT
END;

IClassFactoryVT = POINTER TO RECORD (IUnknownVT)
  CreateInstance: PROCEDURE [stdcall] (this: Interface;
    outerIfc: Interface; iid: GUID;
    VAR ifc: Interface): HRESULT;
  LockServer: PROCEDURE [stdcall] (this: Interface;
    lock: BOOLEAN): HRESULT
END;

IPropertyBagVT = POINTER TO RECORD (IUnknownVT)
  Read: PROCEDURE [stdcall] (this: Interface; name: LONGINT;
    VAR data: VARIANT; errorIfc: Interface): HRESULT;
  RemoteRead: PROCEDURE [stdcall] (this: Interface; name: LONGINT;
    VAR val: VARIANT; type: DWORD; unkIfc: Interface): HRESULT;
  Write: PROCEDURE [stdcall] (this: Interface; name: LONGINT;
    VAR data: VARIANT): HRESULT
END;

Object = POINTER TO ObjDesc;

InterfaceDesc = RECORD
  vtbl: IUnknownVT;
  obj: Object;
  next: Interface;
  iid: GUID
END;

(*object declarations*)

Account = POINTER TO RECORD
  left, right: Account;
  bal, min: LONGINT;
  id: String
END;

ObjDesc = RECORD
  prev, next: Object;
  mcref: LONGINT;
  ifc: Interface;
  eventIfc: Interface;
  bal: LONGINT;
  AB: Account;
  val: String
END;
```

```
VAR str: ARRAY 50 OF CHAR;
  clsidCOMOberon, tlbidCOMOberon, oiidCOMOberon: GUID;
    (*defined by COMOberon*)
  NULLGUID, IIDIUnknown, IIDIDispatch, IIDIClassFactory: GUID;
    (*predefined*)
  IIDIPersistPropertyBag: GUID; (*predefined*)
  IIDIConnectionPointContainer, IIDIConnectionPoint: GUID;
    (*predefined*)
  IUnknownVTImpl: IUnknownVT;
  IDispatchVTImpl: IDispatchVT;
  IClassFactoryVTImpl: IClassFactoryVT;
  IPersistPropertyBagVTImpl: IPersistPropertyBagVT;
  IConnectionPointContainerVTImpl: IConnectionPointContainerVT;
  IConnectionPointVTImpl: IConnectionPointVT;
  class: Object; (*class object*)
  nofServerLocks, nofComponents: LONGINT;
  dispTab: ARRAY MaxFuncs OF String;

  DLL: LONGINT; (*HANDLE*)

(* logging *)

PROCEDURE Log (str: ARRAY OF CHAR);
BEGIN Console.Str(str); Console.Ln
END Log;

PROCEDURE Log2 (str1, str2: ARRAY OF CHAR);
BEGIN Console.Str(str1); Console.Ch(" "); Console.Str(str2); Console.Ln
END Log2;

(* GUID service *)

PROCEDURE EncodeGUID (str: ARRAY OF CHAR; VAR g: GUID);
  VAR i: INTEGER;

  PROCEDURE Val (ch: CHAR): INTEGER;
  BEGIN
    IF ch >= "A" THEN RETURN ORD(ch) + 10 - ORD("A")
      ELSE RETURN ORD(ch) - ORD("0")
    END
  END Val;

  PROCEDURE ToInt (i: INTEGER; VAR n: INTEGER);
    VAR j: INTEGER;
  BEGIN n := Val(str[i]);
    FOR j := 1 TO 3 DO n := n*16 + Val(str[i+j]) END
  END ToInt;

  PROCEDURE ToLong (i: INTEGER; VAR n: LONGINT);
    VAR j: INTEGER;
  BEGIN n := Val(str[i]);
    FOR j := 1 TO 7 DO n := n*16 + Val(str[i+j]) END
  END ToLong;

  PROCEDURE ToStr (i: INTEGER; VAR a: ARRAY OF CHAR; j: INTEGER);
  BEGIN
    WHILE ("A" <= CAP(str[i])) & (CAP(str[i]) <= "F")
      OR ("0" <= str[i]) & (str[i] <= "9") DO
      a[j] := CHR(Val(str[i])*16 + Val(str[i+1])); i := i + 2; INC(j)
    END
  END ToStr;
```

```
BEGIN
  ToLong(1, g.p1); ToInt(10, g.p2); ToInt(15, g.p3);
  ToStr(20, g.p4, 0); ToStr(25, g.p4, 2)
END EncodeGUID;

PROCEDURE DecodeGUID (VAR g: GUID; VAR str: ARRAY OF CHAR);
  VAR i, j: INTEGER;

  PROCEDURE HexDig (n: LONGINT): CHAR;
  BEGIN
    IF n <= 9 THEN RETURN CHR(ORD("0") + n)
      ELSE RETURN CHR(ORD("A") + n - 10)
    END
  END HexDig;

  PROCEDURE IntToHex (n: LONGINT; pos, len: INTEGER);
    VAR i: INTEGER;
  BEGIN i := len;
    WHILE i # 0 DO
      str[pos + i - 1] := HexDig(n MOD 16); n := n DIV 16; DEC(i)
    END
  END IntToHex;

BEGIN str[0] := "{";
  IntToHex(g.p1, 1, 8); str[9] := "-";
  IntToHex(g.p2, 10, 4); str[14] := "-";
  IntToHex(g.p3, 15, 4); str[19] := "-";
  i := 0; j := 20;
  WHILE i # 2 DO
    str[j] := HexDig(ORD(g.p4[i]) DIV 16);
    str[j+1] := HexDig(ORD(g.p4[i]) MOD 16); INC(i); j := j + 2
  END;
  str[24] := "-"; j := 25;
  WHILE i # 8 DO
    str[j] := HexDig(ORD(g.p4[i]) DIV 16);
    str[j+1] := HexDig(ORD(g.p4[i]) MOD 16); INC(i); j := j + 2
  END;
  str[37] := "}"; str[38] := 0X
END DecodeGUID;

PROCEDURE EQUGUID (VAR g1, g2: GUID): BOOLEAN;
  VAR i: INTEGER;
BEGIN i := 0;
  WHILE (i # 8) & (g1.p4[i] = g2.p4[i]) DO INC(i) END;
  RETURN (i = 8) & (g1.p1 = g2.p1) & (g1.p2 = g2.p2) & (g1.p3 = g2.p3)
END EQUGUID;

(* basic COM data type service*)

PROCEDURE HRESULTOK (res: HRESULT): BOOLEAN;
BEGIN RETURN res >= 0
END HRESULTOK;

PROCEDURE DiffSTR (VAR a, b: ARRAY OF CHAR): INTEGER;
  VAR i: INTEGER;
BEGIN i := 0;
  WHILE (a[i] # 0X) & (b[i] # 0X) & (a[i] = b[i]) DO INC(i) END;
  RETURN ORD(a[i]) - ORD(b[i])
END DiffSTR;

PROCEDURE DiffSTRBSTR (VAR a: ARRAY OF CHAR; b: BSTR): INTEGER;
  VAR i: INTEGER;
BEGIN i := 0;
```

```
  WHILE (a[i] # 0X) & (b.str[i] # 0) & (ORD(a[i]) = b.str[i]) DO
    INC(i)
  END;
  RETURN ORD(a[i]) - b.str[i]
END DiffSTRBSTR;

PROCEDURE STRtoBSTR(a: ARRAY OF CHAR; VAR b: BSTR);
  VAR i: INTEGER;
BEGIN i := 0;
  WHILE a[i] # 0X DO b.str[i] := ORD(a[i]); INC(i) END;
  b.str[i] := 0; b.len := i*2
END STRtoBSTR;

PROCEDURE STRtoVARIANT (str: ARRAY OF CHAR; VAR var: VARIANT);
  VAR i: INTEGER; val: BSTR;
BEGIN NEW(val); i := 0;
  WHILE str[i] # 0X DO val.str[i] := ORD(str[i]); INC(i) END;
  val.str[i] := 0; val.len := i*2;
  var.vt := VTBSTR; var.v1 := SYSTEM.VAL(LONGINT, val) + 4
END STRtoVARIANT;

PROCEDURE LONGtoVARIANT (n: LONGINT; VAR var: VARIANT);
BEGIN var.vt := VTI4; var.v1 := n
END LONGtoVARIANT;

PROCEDURE VARIANTtoSTR (VAR var: VARIANT; VAR str: ARRAY OF CHAR);
  VAR i: INTEGER; val: BSTR; vref: VARIANTRef;
BEGIN;
  IF var.vt = VTVARIANT THEN
    vref := SYSTEM.VAL(VARIANTRef, var.v1); var := vref^
  END;
  IF var.vt = VTBSTR THEN
    val := SYSTEM.VAL(BSTR, var.v1 - 4); i := 0;
    WHILE val.str[i] # 0 DO str[i] := CHR(val.str[i]); INC(i) END;
    str[i] := 0X
  ELSE str[0] := 0X (*variant not a string*)
  END
END VARIANTtoSTR;

PROCEDURE VARIANTtoLONG (VAR var: VARIANT; VAR n: LONGINT);
  VAR i: INTEGER; val: BSTR; neg: BOOLEAN; dig: INTEGER;
    vref: VARIANTRef; str: ARRAY 40 OF CHAR;
BEGIN
  IF var.vt = VTVARIANT THEN
    vref := SYSTEM.VAL(VARIANTRef, var.v1); var := vref^
  END;
  IF var.vt = VTI4 THEN n := var.v1
    ELSIF var.vt = VTI2 THEN n := SHORT(var.v1)
    ELSE n := 0 (*variant is not a long number*)
  END
END VARIANTtoLONG;

PROCEDURE VARIANTtoDIFC (VAR var: VARIANT; VAR ifc: Interface);
  VAR h: HANDLE;
BEGIN
  IF var.vt = VTDISPATCH THEN
    h := SYSTEM.VAL(HANDLE, var.v1);
    ifc := SYSTEM.VAL(Interface, h.ref);
  ELSE ifc := NIL (*variant not a dispatch interface*)
  END
END VARIANTtoDIFC;

(* basic COM interface service *)
```

```
PROCEDURE AddObj (obj: Object);
BEGIN obj.next := class.next; class.next := obj;
  obj.next.prev := obj; obj.prev := class
END AddObj;

PROCEDURE RemoveObj (obj: Object);
BEGIN obj.prev.next := obj.next; obj.next.prev := obj.prev
END RemoveObj;

PROCEDURE AddIfc (obj: Object; VAR iid: GUID; vtbl: IUnknownVT);
  VAR ifc: Interface;
BEGIN NEW(ifc); ifc.vtbl := vtbl; ifc.iid := iid;
  ifc.obj := obj; ifc.next := obj.ifc; obj.ifc := ifc
END AddIfc;

(* IUnknown implementation *)

PROCEDURE [stdcall] QueryInterfaceImpl (this: Interface; iid: GUID;
  VAR ifc: Interface): HRESULT;
  VAR str: ARRAY 40 OF CHAR;
BEGIN
  DecodeGUID(iid, str); Log2("QueryInterface", str);
  ifc := this.obj.ifc;
  WHILE (ifc # NIL) & ~EQUGUID(iid, ifc.iid) DO ifc := ifc.next END;
  IF ifc # NIL THEN INC(this.obj.mcref); RETURN SOk
    ELSE RETURN ENoInterface
  END
END QueryInterfaceImpl;

PROCEDURE [stdcall] AddRefImpl (this: Interface): LONGINT;
  VAR str: ARRAY 40 OF CHAR;
BEGIN
  DecodeGUID(this.iid, str); Log2("AddRef", str);
  INC(this.obj.mcref); RETURN this.obj.mcref
END AddRefImpl;

PROCEDURE [stdcall] ReleaseImpl (this: Interface): LONGINT;
  VAR str: ARRAY 40 OF CHAR;
BEGIN
  DecodeGUID(this.iid, str); Log2("Release", str);
  IF this.obj.mcref # 0 THEN DEC(this.obj.mcref);
    IF this.obj.mcref =  0 THEN DEC(nofComponents);
      RemoveObj(this.obj)
    END
  END;
  RETURN this.obj.mcref
END ReleaseImpl;

(* COMOberon dispatched functionality *)

PROCEDURE Invoke (ifc: Interface; dispid: DISPID);
BEGIN
END Invoke;

PROCEDURE Bankrupt (this: Interface; dispIfc: Interface;
  unbal: LONGINT);
  VAR hr: HRESULT; vtbl: IDispatchVT; newIfc: Interface;
    dispid: DISPID; b: BSTR;
    name: LONGINT; h: HANDLE; par: DISPPARAMS; res: VARIANT;
    pArgErr: UINT;
BEGIN
  IF dispIfc # NIL THEN (*try type library interface*)
```

```
    Log("Call QueryInterface");
    hr := dispIfc.vtbl.QueryInterface(dispIfc, oiidCOMOberon, newIfc);
    IF HRESULTOK(hr) THEN
      par.cArgs := 2; par.cnamedArgs := 0;
      NEW(par.rgvarg); par.rgvarg.arg[0].vt := VTI4;
      par.rgvarg.arg[0].v1 := unbal;
      NEW(h); h.ref := SYSTEM.VAL(REFANY, this);
      par.rgvarg.arg[1].vt := VTDISPATCH;
      par.rgvarg.arg[1].v1 := SYSTEM.VAL(LONGINT, h);
      vtbl := SYSTEM.VAL(IDispatchVT, newIfc.vtbl);
      Log("Call Invoke");
      hr := vtbl.Invoke(newIfc, 1, NULLGUID, 0, 1, par, res, NIL,
        pArgErr)
    ELSE (*try class interface*) NEW(b);
      STRtoBSTR("HandleBankrupcy", b);
      name := SYSTEM.VAL(LONGINT, b) + 4;
      vtbl := SYSTEM.VAL(IDispatchVT, dispIfc.vtbl);
      Log("Call GetIDsOfNames");
      hr := vtbl.GetIDsOfNames(dispIfc, NULLGUID, name, 1, 0, dispid);
      IF HRESULTOK(hr) THEN
        par.cArgs := 0; par.cnamedArgs := 0;
        Log("Call Invoke");
        hr := vtbl.Invoke(dispIfc, dispid, NULLGUID, 0, 1, par,
          res, NIL, pArgErr)
      END
    END
  END
END Bankrupt;

PROCEDURE Change (this: Interface; eventIfc: Interface;
  a: Account; amt: LONGINT);
BEGIN
  IF a.bal + amt < a.min THEN amt := a.min - a.bal END;
  IF this.obj.bal + amt < 0 THEN
    Bankrupt(this, eventIfc, this.obj.bal + amt);
    IF this.obj.bal + amt < 0 (*second chance*) THEN
      a.bal := a.bal - this.obj.bal; this.obj.bal := 0
    ELSE a.bal := a.bal + amt; this.obj.bal := this.obj.bal+ amt
    END
  ELSE a.bal := a.bal + amt; this.obj.bal := this.obj.bal+ amt
  END
END Change;

PROCEDURE Search (VAR root: Account; VAR id: ARRAY OF CHAR): Account;
  VAR d: INTEGER;
BEGIN
  IF root = NIL THEN RETURN NIL
  ELSE d := DiffSTR(id, root.id);
    IF d < 0 THEN RETURN Search(root.left, id)
      ELSIF d > 0 THEN RETURN Search(root.right, id)
      ELSE (*d = 0*) RETURN root
    END
  END
END Search;

PROCEDURE Trans (this: Interface; ifc: Interface;
  VAR id: ARRAY OF CHAR;
  amt: LONGINT; VAR a: Account);
BEGIN a := Search(this.obj.AB, id);
  IF a # NIL THEN Change(this, ifc, a, amt) END
END Trans;

PROCEDURE Insert (VAR root, a: Account);
```

```
  VAR d: INTEGER;
BEGIN
  IF root = NIL THEN root := a
  ELSE d := DiffSTR(a.id, root.id);
    IF d < 0 THEN Insert(root.left, a)
      ELSIF d > 0 THEN Insert(root.right, a)
      ELSE (*d = 0*) root.min := a.min
    END
  END
END Insert;

PROCEDURE Open (this, eventIfc: Interface; VAR id: ARRAY OF CHAR;
  bal, min: LONGINT;
  VAR a: Account);
  VAR i, j: INTEGER;
BEGIN NEW(a); i := 0;
  WHILE id[i] # 0X DO a.id[i] := id[i]; INC(i) END;
  a.id[i] := 0X; a.min := min; a.bal := 0; Insert(this.obj.AB, a);
  Change (this, eventIfc, a, bal)
END Open;

(* IDispatch implementation *)

PROCEDURE [stdcall] GetTypeInfoCountImpl (this: Interface;
  VAR tic: UINT): HRESULT;
BEGIN Log("GetTypeInfoCount"); tic := 0; RETURN SOk
END GetTypeInfoCountImpl;

PROCEDURE [stdcall] GetTypeInfoImpl (this: Interface; tic: UINT;
  lcid: LCID;
  VAR tinfo: TYPEINFO): HRESULT;
BEGIN Log("GetTypeInfo"); RETURN EFail
END GetTypeInfoImpl;

PROCEDURE [stdcall] GetIDsOfNamesImpl (this: Interface; iid: GUID;
  VAR name: LONGINT;
  cn: UINT; lcid: LCID; VAR dispid: DISPID): HRESULT;
  VAR i: INTEGER; b: BSTR;
BEGIN
  Log("GetIDsOfNames");
  b := SYSTEM.VAL(BSTR, name - 4); i := 0;
  WHILE (dispTab[i, 0] # 0X) & (DiffSTRBSTR(dispTab[i], b) # 0) DO
    INC(i)
  END;
  IF dispTab[i, 0] # 0X THEN dispid := i; RETURN SOk
    ELSE RETURN EFail
  END
END GetIDsOfNamesImpl;

PROCEDURE [stdcall] InvokeImpl (this: Interface; dispid: DISPID;
  iid: GUID; lcid: LCID;
  wFlags: WORD; VAR par: DISPPARAMS; VAR res: VARIANT;
    excpInfo: EXCEPINFO;
  VAR pArgErr: UINT): HRESULT;
  VAR i: INTEGER; a: Account; amt, min: LONGINT; id: String;
  eventIfc: Interface;
BEGIN
  CASE dispid OF
    0: IF ODD(wFlags DIV 2) THEN Log("Invoke Value Get");
          (*DISPATCHPROPERTYGET*)
          STRtoVARIANT(this.obj.val, res); RETURN SOk
       ELSIF ODD(wFlags DIV 4) THEN Log("Invoke Value Put");
          (*DISPLAYPROPERTYPUT*)
```

```
      VARIANTtoSTR(par.rgvarg.arg[0], this.obj.val); RETURN SOk
    ELSE RETURN EFail
    END |
1: Log("Invoke Open");
    IF ODD(wFlags) THEN (*DISPATCHMETHOD*) i := 0;
      WHILE par.cArgs > 4 DO DEC(par.cArgs); INC(i) END;
      IF par.cArgs = 4 THEN
        VARIANTtoDIFC(par.rgvarg.arg[i], eventIfc);
        DEC(par.cArgs); INC(i)
      ELSE eventIfc := NIL
      END;
      IF par.cArgs = 3 THEN
        VARIANTtoLONG(par.rgvarg.arg[i], min); DEC(par.cArgs);
        INC(i)
      ELSE min := 0
      END;
      IF par.cArgs = 2 THEN VARIANTtoLONG(par.rgvarg.arg[i], amt);
        DEC(par.cArgs); INC(i)
      ELSE amt := 0
      END;
      IF par.cArgs = 1 THEN VARIANTtoSTR(par.rgvarg.arg[i], id);
        DEC(par.cArgs); INC(i)
      ELSE id := "OWN"
      END;
      Open(this, eventIfc, id, amt, min, a);
      LONGtoVARIANT(a.bal, res);
      RETURN SOk
    ELSE RETURN EFail
    END |
2: Log("Invoke Query");
    IF ODD(wFlags) THEN (*DISPATCHMETHOD*) i := 0;
      WHILE par.cArgs > 1 DO DEC(par.cArgs); INC(i) END;
      IF par.cArgs = 1 THEN
        VARIANTtoSTR(par.rgvarg.arg[i], id); DEC(par.cArgs); INC(i)
      ELSE id := "OWN"
      END;
      a := Search(this.obj.AB, id);
      IF a # NIL THEN LONGtoVARIANT(a.bal, res)
        ELSE STRtoVARIANT("unknown", res)
      END;
      RETURN SOk
    ELSE RETURN EFail
    END |
3: Log("Trans");
    IF ODD(wFlags) THEN (*DISPATCHMETHOD*) i := 0;
      WHILE par.cArgs > 3 DO DEC(par.cArgs); INC(i) END;
      IF par.cArgs = 3 THEN
        VARIANTtoDIFC(par.rgvarg.arg[i], eventIfc);
        DEC(par.cArgs); INC(i)
      ELSE eventIfc := NIL
      END;
      IF par.cArgs = 2 THEN
        VARIANTtoLONG(par.rgvarg.arg[i], amt); DEC(par.cArgs);
        INC(i)
      ELSE amt := 0
      END;
      IF par.cArgs = 1 THEN
        VARIANTtoSTR(par.rgvarg.arg[i], id); DEC(par.cArgs); INC(i)
      ELSE id := "OWN"
      END;
      Trans(this, eventIfc, id, amt, a);
      IF a # NIL THEN LONGtoVARIANT(a.bal, res)
        ELSE STRtoVARIANT("unknown", res)
```

```
          END;
          RETURN SOk
        ELSE RETURN EFail
        END
    ELSE RETURN EFail
    END
  END InvokeImpl;

(* IConnectionPointContainer implementation *)

PROCEDURE [stdcall] EnumConnectionPointsImpl (this: Interface;
  VAR ifc: Interface): HRESULT;
BEGIN Log("EnumConnectionPoints"); ifc := NIL; RETURN EFail
END EnumConnectionPointsImpl;

PROCEDURE [stdcall] FindConnectionPointImpl (this: Interface;
  iid: GUID;
  VAR ifc: Interface): HRESULT;
  VAR str: ARRAY 40 OF CHAR;
BEGIN
  DecodeGUID(iid, str); Log2("FindConnectionPoint", str);
  IF EQUGUID(iid, oiidCOMOberon) THEN ifc := this.obj.ifc;
    WHILE (ifc # NIL) & ~EQUGUID(IIDIConnectionPoint, ifc.iid) DO
      ifc := ifc.next
    END;
    IF ifc # NIL THEN RETURN SOk ELSE RETURN ENoInterface END
  END
END FindConnectionPointImpl;

(* IConnectionPoint implementation *)

PROCEDURE [stdcall] GetConnectionInterfaceImpl (this: Interface;
  VAR iid: GUID): HRESULT;
BEGIN Log("GetConnectionInterface"); iid := oiidCOMOberon; RETURN SOk
END GetConnectionInterfaceImpl;

PROCEDURE [stdcall] GetConnectionPointContainerImpl (this: Interface;
  VAR ifc: Interface): HRESULT;
BEGIN
  Log("GetConnectionPointContainer"); ifc := this.obj.ifc;
  WHILE (ifc # NIL) & ~EQUGUID(IIDIConnectionPointContainer, ifc.iid)
    DO ifc := ifc.next
  END;
  RETURN SOk
END GetConnectionPointContainerImpl;

PROCEDURE [stdcall] AdviseImpl (this, ifc: Interface;
  VAR key: DWORD): HRESULT;
BEGIN Log("Advise"); this.obj.eventIfc := ifc; key := 1; RETURN SOk
END AdviseImpl;

PROCEDURE [stdcall] UnadviseImpl (this: Interface;
  key: DWORD): HRESULT;
BEGIN Log("Unadvise");
  IF key = 1 THEN this.obj.eventIfc := NIL; RETURN SOk
    ELSE RETURN EFail
  END
END UnadviseImpl;

PROCEDURE [stdcall] EnumConnectionsImpl (this: Interface;
  VAR ifc: Interface): HRESULT;
BEGIN Log("EnumConnections"); ifc := NIL; RETURN EFail
END EnumConnectionsImpl;
```

```
(* IPersist implementation *)

PROCEDURE [stdcall] GetClassIDImpl (this: Interface;
  VAR clsid: GUID): HRESULT;
BEGIN Log("GetClassID"); RETURN EFail
END GetClassIDImpl;

(* IPersistPropertyBag implementation *)

PROCEDURE [stdcall] InitNewImpl (this: Interface): HRESULT;
BEGIN Log("InitNew");
  this.obj.val := "Any E-Bank"; NEW(this.obj.AB); RETURN SOk
END InitNewImpl;

PROCEDURE [stdcall] LoadImpl (this, bagIfc,
  errorIfc: Interface): HRESULT;
  VAR var: VARIANT; hr: HRESULT; b: BSTR; vtbl: IPropertyBagVT;
BEGIN Log("Load");
  NEW(b); STRtoBSTR(dispTab[0], b);
  vtbl := SYSTEM.VAL(IPropertyBagVT, bagIfc.vtbl);
  hr := vtbl.Read(bagIfc,  SYSTEM.VAL(LONGINT, b) + 4, var, errorIfc);
  IF HRESULTOK(hr) THEN
    VARIANTtoSTR(var, this.obj.val); NEW(this.obj.AB) END;
  RETURN hr
END LoadImpl;

PROCEDURE [stdcall] SaveImpl (this: Interface;
  bagIfc: Interface; clearDirty,
  saveAll: BOOLEAN): HRESULT;
  VAR var: VARIANT; b: BSTR; vtbl: IPropertyBagVT;
BEGIN Log("Save");
  NEW(b); STRtoBSTR(dispTab[0], b); STRtoVARIANT(this.obj.val, var);
  vtbl := SYSTEM.VAL(IPropertyBagVT, bagIfc.vtbl);
  RETURN vtbl.Write(bagIfc, SYSTEM.VAL(LONGINT, b) + 4, var)
END SaveImpl;

(* IClassFactory implementation *)

PROCEDURE [stdcall] CreateInstanceImpl (this: Interface;
  outerIfc: Interface; iid: GUID;
  VAR ifc: Interface): HRESULT;
  VAR obj: Object;
BEGIN Log("CreateInstance");
  IF outerIfc = NIL THEN NEW(obj);
    AddIfc(obj, IIDIPersistPropertyBag, IPersistPropertyBagVTImpl);
    AddIfc(obj, IIDIConnectionPointContainer,
      IConnectionPointContainerVTImpl);
    AddIfc(obj, IIDIDispatch, IDispatchVTImpl);
    AddIfc(obj, IIDIUnknown, IUnknownVTImpl);
    ifc := obj.ifc;
    WHILE (ifc # NIL) & ~EQUGUID(iid, ifc.iid) DO ifc := ifc.next END;
    IF ifc # NIL THEN
      obj.mcref := 1; AddObj(obj); INC(nofComponents); RETURN SOk
    ELSE RETURN ENoInterface
    END
  ELSE ifc := NIL; RETURN ClassENoAggregation
  END
END CreateInstanceImpl;

PROCEDURE [stdcall] LockServerImpl (this: Interface;
  lock: BOOLEAN): HRESULT;
  VAR res: LONGINT;
```

```
BEGIN Log("LockServer");
  IF lock THEN res := Kernel32.InterlockedIncrement(nofServerLocks)
    ELSE res := Kernel32.InterlockedDecrement(nofServerLocks)
  END
END LockServerImpl;

(* DLL interface and implementation*)

PROCEDURE [stdcall] DllCanUnloadNow* (): HRESULT;
BEGIN Log("DllCanUnloadNow");
  IF (nofComponents = 0) & (nofServerLocks = 0) THEN RETURN SOk
    ELSE RETURN SFalse
  END
END DllCanUnloadNow;

PROCEDURE [stdcall] DllGetClassObject* (clsid, iid: GUID;
  VAR ifc: Interface): HRESULT;
BEGIN Log("DllGetClassObject");
  IF EQUGUID(clsid, clsidCOMOberon) THEN ifc := class.ifc;
    WHILE (ifc # NIL) & ~EQUGUID(iid, ifc.iid) DO ifc := ifc.next END;
    IF ifc # NIL THEN RETURN SOk ELSE RETURN ENoInterface END
  ELSE RETURN ClassEClassNotAvailable
  END
END DllGetClassObject;

PROCEDURE Append (VAR key: ARRAY OF CHAR; pos: INTEGER;
  s: ARRAY OF CHAR);
  VAR i, j: INTEGER;
BEGIN i := pos; j := 0;
  WHILE s[j] # 0X DO key[i] := s[j]; INC(i); INC(j) END;
  key[i] := 0X
END Append;

PROCEDURE Register (VAR key: ARRAY OF CHAR; val: ARRAY OF CHAR);
BEGIN Registry.SetKeyValue(Registry.ClassesRoot, key, "", val);
  Console.Str(key); Console.Str(" = "); Console.Str(val); Console.Ln
END Register;

PROCEDURE [stdcall] DllRegisterServer* (): HRESULT;
  VAR res: LONGINT; DLLName: String; key: ARRAY 80 OF CHAR;
BEGIN Log("DllRegisterServer");
  res := Kernel32.GetModuleFileName(Kernel.hInstance, DLLName, MaxLen);
  Append(key, 0, "CLSID\"); Append(key, 6, CLSIDCOMOberon);
  Register(key, ObjectCaption);
  Append(key, 44, "\InprocServer32"); Register(key, DLLName);
  Append(key, 44, "\ProgID"); Register(key, ProgID);
  Append(key, 44, "\VersionIndependentProgID");
  Register(key, VIndProgID);
  Append(key, 44, "\Insertable"); Register(key, "");
  Append(key, 44, "\Control"); Register(key, "");
  Append(key, 44, "\ToolboxBitmap32"); Register(key, BitmapID);
  Append(key, 44, "\Version"); Register(key, VersionNo);
  Append(key, 44, "\TypeLib"); Register(key, TLBIDCOMOberon);
  Append(key, 0, VIndProgID); Register(key, ObjectCaption);
  Append(key, 9, "\CLSID"); Register(key, CLSIDCOMOberon);
  Append(key, 9, "\CurVer"); Register(key, ProgID);
  Append(key, 0, ProgID); Register(key, ObjectCaption);
  Append(key, 11, "\CLSID"); Register(key, CLSIDCOMOberon);
  Append(key, 0, "TypeLib\"); Append(key, 8, TLBIDCOMOberon);
   Register(key, "");
  Append(key, 46, "\1.0"); Register(key, "");
  Append(key, 50, "\0"); Register(key, "");
  Append(key, 52, "\Win32"); Register(key, TLBPathName);
```

```
    Append(key, 0, "Interface\"); Append(key, 10, OIIDCOMOberon);
    Register(key, InterfaceCaption);
    Append(key, 48, "\TypeLib"); Register(key, TLBIDCOMOberon);
    RETURN SOk
  END DllRegisterServer;

  PROCEDURE Unregister (key: ARRAY OF CHAR);
  BEGIN Registry.DeletePath(Registry.ClassesRoot, key);
    Console.Str(key); Console.Str(" unregistered"); Console.Ln
  END Unregister;

  PROCEDURE [stdcall] DllUnregisterServer* (): HRESULT;
    VAR key: ARRAY 80 OF CHAR;
  BEGIN Log("DllUnregisterServer");
    Append(key, 0, "CLSID\"); Append(key, 6, CLSIDCOMOberon);
    Unregister(key);
    Append(key, 0, "TypeLib\"); Append(key, 8, TLBIDCOMOberon);
    Unregister(key);
    Unregister(VIndProgID);
    Unregister(ProgID);
    RETURN SOk
  END DllUnregisterServer;

  PROCEDURE InitAPI ();
  BEGIN DLL := Kernel32.LoadLibrary("OLE32.DLL")
  END InitAPI;

  PROCEDURE* FreeAPI ();
    VAR done: BOOLEAN;
  BEGIN done := Kernel32.FreeLibrary(DLL); DLL := 0
  END FreeAPI;

BEGIN
  EncodeGUID("{00000000-0000-0000-0000-000000000000}", NULLGUID);
  EncodeGUID("{00000000-0000-0000-C000-000000000046}", IIDIUnknown);
  EncodeGUID("{00020400-0000-0000-C000-000000000046}", IIDIDispatch);
  EncodeGUID("{00000001-0000-0000-C000-000000000046}", IIDIClassFactory);
  EncodeGUID("{37D84F60-42CB-11CE-8135-00AA004BB851}",
    IIDIPersistPropertyBag);
  EncodeGUID("{B196B284-BAB4-101A-B69C-00AA00341D07}",
    IIDIConnectionPointContainer);
  EncodeGUID("{B196B286-BAB4-101A-B69C-00AA00341D07}",
    IIDIConnectionPoint);
  EncodeGUID(CLSIDCOMOberon, clsidCOMOberon);
  EncodeGUID(TLBIDCOMOberon, tlbidCOMOberon);
  EncodeGUID(OIIDCOMOberon, oiidCOMOberon);
  dispTab[0] := "Value"; dispTab[1] := "Open";
  dispTab[2] := "Query"; dispTab[3] := "Trans";
  NEW(IUnknownVTImpl);
  IUnknownVTImpl.QueryInterface := QueryInterfaceImpl;
  IUnknownVTImpl.AddRef := AddRefImpl;
  IUnknownVTImpl.Release := ReleaseImpl;
  NEW(IDispatchVTImpl);
  IDispatchVTImpl.QueryInterface := QueryInterfaceImpl;
  IDispatchVTImpl.AddRef := AddRefImpl;
  IDispatchVTImpl.Release := ReleaseImpl;
  IDispatchVTImpl.GetTypeInfoCount := GetTypeInfoCountImpl;
  IDispatchVTImpl.GetTypeInfo := GetTypeInfoImpl;
  IDispatchVTImpl.GetIDsOfNames := GetIDsOfNamesImpl;
  IDispatchVTImpl.Invoke := InvokeImpl;
  NEW(IClassFactoryVTImpl);
  IClassFactoryVTImpl.QueryInterface := QueryInterfaceImpl;
  IClassFactoryVTImpl.AddRef := AddRefImpl;
```

```
  IClassFactoryVTImpl.Release := ReleaseImpl;
  IClassFactoryVTImpl.CreateInstance := CreateInstanceImpl;
  IClassFactoryVTImpl.LockServer := LockServerImpl;
  NEW(IConnectionPointContainerVTImpl);
  IConnectionPointContainerVTImpl.QueryInterface := QueryInterfaceImpl;
  IConnectionPointContainerVTImpl.AddRef := AddRefImpl;
  IConnectionPointContainerVTImpl.Release := ReleaseImpl;
  IConnectionPointContainerVTImpl.EnumConnectionPoints :=
    EnumConnectionPointsImpl;
  IConnectionPointContainerVTImpl.FindConnectionPoint :=
    FindConnectionPointImpl;
  NEW(IConnectionPointVTImpl);
  IConnectionPointVTImpl.QueryInterface := QueryInterfaceImpl;
  IConnectionPointVTImpl.AddRef := AddRefImpl;
  IConnectionPointVTImpl.Release := ReleaseImpl;
  IConnectionPointVTImpl.GetConnectionInterface :=
    GetConnectionInterfaceImpl;
  IConnectionPointVTImpl.GetConnectionPointContainer :=
    GetConnectionPointContainerImpl;
  IConnectionPointVTImpl.Advise := AdviseImpl;
  IConnectionPointVTImpl.Unadvise := UnadviseImpl;
  IConnectionPointVTImpl.EnumConnections := EnumConnectionsImpl;
  NEW(IPersistPropertyBagVTImpl);
  IPersistPropertyBagVTImpl.QueryInterface := QueryInterfaceImpl;
  IPersistPropertyBagVTImpl.AddRef := AddRefImpl;
  IPersistPropertyBagVTImpl.Release := ReleaseImpl;
  IPersistPropertyBagVTImpl.GetClassID := GetClassIDImpl;
  IPersistPropertyBagVTImpl.InitNew := InitNewImpl;
  IPersistPropertyBagVTImpl.Load := LoadImpl;
  IPersistPropertyBagVTImpl.Save := SaveImpl;
  NEW(class);
  AddIfc(class, IIDIClassFactory, IClassFactoryVTImpl);
  AddIfc(class, IIDIUnknown, IUnknownVTImpl);
  class.mcref := 1; class.prev := class; class.next := class;
  InitAPI(); (*based on DllMain in Kernel32*)
  Modules.InstallTermHandler(FreeAPI)
END COMOberon.
```