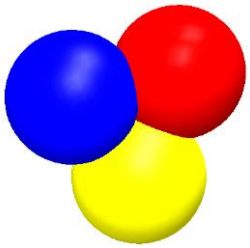


ObjAsm

x86 / x64

Object Oriented Programming in Assembly

1. Introduction

ObjAsm is a MASM® compatible single inheritance object oriented implementation for x86 and x64 capable hardware.

It makes easy to perform complex tasks and to write large projects using very efficient assembly code in a more productive way. It has an extensive, optimized, and tested object repository and procedure library that can be used immediately in your applications.

Object Oriented Programming (OOP) is a programming paradigm that has become the major component of languages with a high level of abstraction such as C++, Pascal or Python just to mention a few. The main goal of using an OOP paradigm in your applications is to encapsulate the data and code that processes that data in a single entity, called object. The data and associated code are collectively referred to as object members.

Some key points about OOP paradigm are:

- ✓ The code in an object usually performs some operation on its own data. This code consists of procedures called methods.
- ✓ The data inside an object is considered private. Only the object's code should directly access its data. In assembly you can easily break this rule but at your own risk. This data can be of any type, even other objects.
- ✓ To use an object, you must first create it in memory. This process is called instantiation and the created object is an instance. You can create as many instances in your application, each with their own data. If the objects are no longer needed, you can destroy them releasing the memory for other uses.
- ✓ You can design new objects based on existing objects. This technique is called inheritance. It allows the reuse of a known and proven code. Using inheritance, you can write an entire application by simply modifying or extending existing object members.

The **ObjAsm** core is a collection of macros and procedures that implement the OOP model with negligible overhead.

2. Contents

3. EULA

END-USER SOFTWARE LICENSE AGREEMENT FOR OBJASM

I**MPORTANT! READ CAREFULLY**: This End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and me for the **ObjAsm** software product accompanying this EULA, which includes computer software and may include "online" or electronic documentation, associated media, and printed materials ("SOFTWARE PRODUCT"). By installing, copying, or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, do not install or use the SOFTWARE PRODUCT.

GRANT OF LICENSE

1. Subjected to the terms and provisions of this Agreement, I grant to you the non-exclusive, limited and royalty-free right to use this Software. The term "Software" includes all elements of the Software such as data files and screen displays. You are not receiving any ownership or proprietary right, title or interest in or to the Software or the copyright, trademarks, or other rights related thereto.

2. You are completely free to distribute your own developed source, object or executable code for commercial purposes, but NOT the source code of the SOFTWARE PRODUCT.

LIMITED WARRANTY

NO WARRANTIES. I expressly disclaim any warranty for the Software Product. THE SOFTWARE PRODUCT AND ANY RELATED DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES AND CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE PRODUCT REMAINS WITH YOU.

NO LIABILITY FOR DAMAGES. In no event shall I be liable for any damages whatsoever (including, without limitation, damages for loss of business profit, business interruption, loss of business information, or any other pecuniary loss) arising out of the use of or inability to use this product, even if it has been advised of the possibility of such damages.

4. Acknowledgements

I would like to express my very great appreciation to

Erica Friedrich Heineken,
Jaymeson Trudgen,
Konstantin Usanov,
Homer,
Kai Liebenau
Héctor S. Enrique

and to all other authors mentioned in the source code whose valuable and constructive contributions made this work possible. Thank you!

Corrections, comments, suggestions, contributions, etc. may be sent to the MASM32 Forum, or directly mailed to:

ObjAsm@gmx.net

G. Friedrich,

January, 2019

5. Abbreviations

BNC: Bitness Neutral Code
CLSID: Class Identifier
COM: Component Object Model
DLL: Dynamic Link Library
DMT: Dual Method Table
EVT: Event
ETT: Event Translation Table
GUID: Globally Unique Identifier
HLL: High Level Language
ID: Identifier
IID: Interface Identifier
IMT: Interface Method Table
Inst: Instance
Mtd: Method
OMD: Object Metadata
OO: Object Oriented
OOP: Object Oriented Programming
sIID: Symbolic Interface Identifier
TPL: Object Template
VMT: Virtual Method Table
ZTC: Zero Terminating Character

6. Basic Concepts

Like other technologies, OOP comes with its own concepts and terminology. You will find many of them in the following paragraphs:

Ancestor object

An object that is the parent of a derived object. The latter inherits all properties of his ancestor. Multiple derived objects can inherit the properties of a common ancestor object. In order to meet their own needs, new specific variables and methods can be added to these objects.

Constructor

A special method that initializes an object instance. In some HLLs like C++, constructors are called automatically. In **ObjAsm**, an object can have more than one constructor and it is the coder responsibility to call it. Because **ObjAsm** initializes object variables automatically, a constructor may not be required. During regular instantiation, the initial values are copied from a template in memory defined at compile time. More on this topic later (see lifecycle).

Derived object

An object that inherits the members (variables and code) of a predecessor object. A derived object can be used as the ancestor from which other objects can be derived. The chain of ancestors and derived objects in an OO program creates a hierarchy of related objects.

Using **ObjAsm**, a derived object inherits the properties of only one ancestor. This is called “single inheritance”. See embedding objects.

Destructor

A special method is called when an object instance is to be destroyed using the *Destroy* or *Kill* macros. This method is responsible for the entire housekeeping, such as the deallocation of the resources usually committed by the constructor.

In **ObjAsm** there is only one automatic destructor called *Done*. It must be redefined for every object that needs to handle situations such as the release of particular resources. *Done* is virtual method, whose address is located in the first placeholder in the virtual method table (VMT).

Encapsulation

The process of linking data and code in an object. A method usually performs some operations on the encapsulated data. Encapsulation limits the use of this data to a predefined set of methods, which simplifies debugging, maintenance and revision.

Inheritance

The process of transferring members of an ancestor to a derived object. By using inheritance, you can quickly improve existing objects by simply modifying existing methods or adding new variables and methods that enrich them.

Instance

Memory used to implement an object.

Member

Any component of an object. Members are methods, variables and other embedded objects.

Method

Another term for an object procedure. A method can be *static*, *virtual*, *dynamic*, *inline* or form part of an *interface*.

Methods can be *private* to limit their visibility (scope) from other objects or modules.

Interface methods are virtual methods used to communicate using the COM technology. The special design of the method jump table preserves the structure of COM interfaces separating them from the rest of the virtual methods. The whole jump table is called Dual Method Table (DMT). COM methods build up the Interface Method Table (IMT) and the remaining regular virtual methods the Virtual Method Table (VMT). Since all object instances of the same type share the same DMT, a change of a method address will affect all instances. This is called object *superclassing*.

While virtual and interface method addresses are stored in the DMT, a dynamic method address is stored as an instance data. This is completely transparent to the coder and allows redefinition of these type of methods at runtime independently on a per instance basis. This is called object *subclassing*.

Object

A structure that relates data and code. It is important to understand that an object is merely a source code description of related data and code. To use an object in a program, you must create an instance of it. The object concept is equivalent to a class in some HLLs like C++ or Pascal.

Polymorphism

The process by which an object can determine the method to execute or the variable to take.

For example, you have two objects, “Triangle” and “Rectangle” derived from a common ancestor called “Shape” and this ancestor defines a *Draw* method. This method is redefined on each of the derived objects to render the corresponding shape. Now it is possible to call the *draw* method regardless of the object type. Ty simply calling the “Shape” *Draw* method of any of the derived object instances, the correct shape rendering procedure is invoked.

The same concept applies to all object members.

Composition

The composition relationship is a part-whole relationship where a part can only be a part of one object at a time. This means that the part is created when the object is created and destroyed when the object is destroyed. To qualify as a composition, the object and a part must have the following relationship

1. The part (member) is part of the object (class).
2. The part (member) can only belong to one object (class).
3. The part (member) has its existence managed by the object (class).
4. The part (member) does not know about the existence of the object (class).

A composition may avoid creating some parts until they are needed.

Using object composition can provide the following benefits:

1. Reuse existing code: object composition allows to reuse of the existing code without a need to model an is-a relationship as usually done in inheritance.
2. Change in implementation of the class used in the composition without requiring external clients: composition also allows to make code easier to change and adapt if necessary. The internal classes can be changed without any side effects and changes can be handled internally.
3. Complex Objects: composition makes handling of complex objects much easier than using multiple inheritance.

ObjAsm implements composition using the Embed macro.

Single inheritance

The technique of building a derived object from a single ancestor object. Some HLLs support the inheritance of more than one ancestor object. This is called multiple inheritance and is not supported by **ObjAsm**.

ObjAsm supports embedding objects, which makes handling complex objects much easier than multiple inheritance.

Virtual method

An object method whose address is defined at compile time and stored in the virtual method table (VMT). This method address is shared with all instances of the same type.

Private method

A method with limited scope that restricts its invocation from within methods of the same object.

Interface method

A virtual method that can be used to implement COM or similar interfaces.

Dynamic method

A method whose address is stored as an object variable and can be redefined for each instance independently.

Static method

A method whose address is resolved at compile time and cannot be changed. The method address is hardcoded. The call of as static method is the fastest invocation type at the cost of polymorphism.

Inline method

A method whose code is expanded in place of the invocation. It can be thought as a MASM® macro that expands where the invocation is done.

7. Using the model

To use **ObjAsm** a few rules must be followed. In the following lines, they are explained in detail.

System setup

All **ObjAsm** applications start including the *Model.inc* file and calling the *SysSetup* macro. Here is where the framework is customized to support a user interface or not, the application bitness, the debugging support and so on.

The *SysInit* macro should be the first code sequence to execute. It initializes all framework runtime variables. Similarly, the last executed code sequence should be the *SysDone* macro just before the ExitProcess API is called. This macro releases all resources assigned by the framework.

Object definition

To declare an object, the *Object* macro is used, as shown in the following example from Demo01.inc file:

```
21  Object Shape, ShapeID, OA:Primer
22    VirtualAbstract     GetArea
23    RedefineMethod   Init, DWORD, DWORD
24
25    DefineVariable   dBase,    DWORD, 0
26    DefineVariable   dHeight, DWORD, 0
27  ObjectEnd
```

The first line (21) starts an object definition called *Shape*. An ID (DWORD) defines its type, which is in this case ShapeID (i.e. 2). Finally, after the type ID, the ancestor object from which this object is derived is declared. In this particular case, OA:Primer defines a destructor called *Done*, that does not perform any operation, but ensures that the first position in the virtual method table (VMT) of all Primer inherited objects is always filled with this method. To implement some action, *Done* must be overridden.

Note: OA:Primer, OA is the declaration of the object namespace used to identify the Primer object.

The next line (22) defines a virtual abstract method called *GetArea*. An abstract method is a placeholder that has to be overwritten in descendant objects to be called. Line 23 defines a virtual method, called *Init*, which will be used as a constructor to initialize the object data. It has two parameters of DWORD type.

Line 25 and 26 define two object variables called *dBase* and *dHeight*, both of DWORD type and initialized to zero.

Object instantiation

ObjAsm can create and setup object instances in different ways. There are two types of instances, dynamic and static.

Dynamic instances

The most common case is the dynamic allocation of the object instance. It is started with the *New* and ended with the *Destroy* macro. *New* allocates memory from the process heap using the *MemAlloc* macro. This memory is immediately initialized copying the data from a template stored in the data segment. The information contained in this template is generated at compile time using the object definition. This ensures that all instances are created with consistent data that the constructor can work with. After the object has completed his purpose, the destructor must be called to release the resources allocated by the constructor. Finally the instance memory must be freed again. These last steps are done by the *Destroy* macro, which automatically calls the default destructor (*Done*) and releases the instance memory using the *MemFree* macro.

```
New Shape
.if eax != NULL
  mov esi, eax
  OCall esi::Shape.Init, 10, 15
  .if eax == OBJ_OK
  …
  Destroy esi
.endif
```

Note: *New* returns a pointer to the object instance. Depending on the target bitness setting, this vale is returned in the rax or eax register.

An object instance can also be placed on the memory stack like a local variable. In this case, the *New* macro can also be used, but it skips the allocation process and continues with the memory initialization from the object template. When the method or procedure is about to return, the epilogue code frees the stack where the object instance resides. This means that the *Destroy* macro should never be used in these situations. On the other hand, the destructor must be called explicitly to free allocated resources.

```
Method …
  local AboutDlg:$Obj(DialogAbout)
  …
  New AboutDlg::DialogAbout
  OCall AboutDlg::DialogAbout.Init, xsi, [xsi].hWnd, NULL, offset szAboutText
  OCall AboutDlg::DialogAbout.Show
  OCall AboutDlg::DialogAbout.Done
  …
```

Static instances

A static instance is memory reserved in the .data or .data? segment, like a global variable.

When the object instance is placed on the .data segment, the $ObjInst macro is used to create a copy of the object template at compile time.

```
.data
MyShape $ObjInst(Shape)
```

When the object instance is placed on the .data? segment, only memory reservation is required and the *New* macro cares at runtime for the initialization.

```
.data?
MyShape $Obj(Shape)  {}

.code
New MyShape::Shape
…
```

In these last two cases, as with instances on the stack, only the destructor has to be called when the object is no longer needed.

The object template, from which objects instances get their initialization data, can also be considered as a static instance. This is especially interesting if we have only a single object instance in the application. In this case, we can use the object template as a static instance. Usually this is the case when the *SdiApp*, *MdiApp*, *DlgApp* or one of its descendant objects are used as the main application.

The object template can be accessed using the *$ObjTmpl()* macro. To call a method of the object template, the following code can be used:

```
OCall $ObjTmpl(Shape)::Shape.Init
```

Note: some objects destroy themselves when the OS destroy the window object associated with them. This behaviour works only on dynamic instances. For this reason, static instances such as SdiApp, MdiApp, DlgApp or their descendants must be explicitly destroyed.

Namespaces

Namespaces were introduced to avoid name clashes between API definitions or third party modules. Namespaces can be used for object definitions. Interfaces and the VMT of the DMT are not affected.

Object members names are automatically mangled to avoid such conflicts. The same applies to interface members.

The regular syntax to reference object is:

```
Instance::Namespace:Object.Method Parameters
```

In some cases, parts of the reference can be omitted and default values are used. I.e. if Instance is omitted inside a method, the *pSelf* value is used. The same way, if the object name is omitted, the name is inferred from the method definition where the reference is used. See simplified syntax.

If not specified, the namespace is the default, called “OA”. To use another namespace use the *ObjNamespace* macro.

Adding objects to the application

There are three ways to add a new object to the application.

1. Using the *MakeObjects* macro
2. Using the *LoadObjects* macro
3. Using the *include* statement

Using the *MakeObjects* macro, the whole file that defined the object is included into the project. The object definition is created and all the methods are compiled.

Using the *LoadObjects* macro, the object must be precompiled beforehand into a static library. A tool like *OATools* can be used for that purpose. This technique has the advantage of improving compilation speed.

Using the *include* statement, the assembler adds the whole file to the application. It has the same effect as using the *MakeObjects* macro, but compilation switches can be set manually. This is useful for object compilation files. See the .asm files in the “Code\Objects” directory.

The *MakeObjects* and *LoadObjects* macros accept multiple arguments for processing many objects in the same call. This way it is possible to write all objects of an inheritance chain in one single line.

Object static library

To improve the application compilation speed, it is possible to precompile object methods into a static library (.lib file). The OATools application can be used for that purpose.

When these precompiled object methods are loaded, we have to avoid a redefinition with their original code. Since the object definition is always required, a conditional compilation switch between the object definition and the object's methods is added using the IMPLEMENT symbol.

Example:

```
Object MyObject,, Primer
 VirtualMethod  MyMethod,  dword
ObjectEnd
```

➔ if IMPLEMENT

```
Method MyObject.MyMethod, uses esi, MyParam:DWORD
 SetObject esi
 ...
MethodEnd
```

➔ endif

An .asm file has to be created to compile each object, indicating to the assembler the complete code dependency, all ancestor objects and switches.

Object variables

Object members used to store values are called object variables. These variables can be of any type known by MASM® like DWORD, WORD or BYTE or any custom type like structures. The *DefineVariable* macro is used to declare a variable, its type and initial value.

Example:

```
Object MyObject,, Primer
 DefineVariabbe  MyVar,  dword,  100
ObjectEnd
```

or if you don't care about the initial value

```
Object MyObject,, Primer
  DefineVariabbe  MyVar,  DWORD,  ?
ObjectEnd
```

You can also use structures as variables. Example:

```
Data struc
  var1   DWORD   ?
  var2   DWORD   ?
  var3   DWORD   ?
Data ends

Object MyObject,, Primer
  DefineVariable  MyData,  Data,  {0,100,20}
ObjectEnd
```

or

```
Object MyObject,, Primer
  DefineVariable  MyData,  Data,  {}
ObjectEnd
```

In this case, the structure variables remain uninitialized.

It is allowed to use the MASM® *DUP* operator using the following syntax:

```
Object MyObject,, Primer
  DefineVariable  MyVar,  dword,  20 DUP(0)
ObjectEnd
```

or

```
Object MyObject,, Primer
  DefineVariable  MyVar,  dword,  20 DUP(?)
ObjectEnd
```

Only the initial variable value can be changed in a descendant object using the *RedefineVariable* macro. Example:

```
Object MyDescendant,, MyObject
  RedefineVariable  MyVar,  250
ObjectEnd
```

Note: that the variable type cannot the modified!

Method definitions

A method definition follows the same rules for MASM® procedures. They begin with the *Method* macro invocation followed by the method name, the procedure modifiers like the calling convention and the uses clauses and finally the argument list. This has to correspond exactly with the arguments declared in the object declaration.

The method code is finalized by the *MethodEnd* macro, which contains an implicit return instruction.

The *Method* macro hides the instance pointer called *pSelf*, which is always passed as first argument of the call. To assign this pointer to a register, the *SetObject* macro has to be used. To finally unassign it, the *ReleaseObject* macro has to be used. If it was not done before, the *MethodEnd* macro automatically calls *ReleaseObject*.

The *MethodEnd* macro also contains a label called *@@EOM* (End Of Method), which simplifies the method exit call. A handy macro called *ExitMethod* was defined to jump to this label. A conditional jump is possible too, adding the jump condition to the macro invocation, i.e.:

```
ExitMethod .if eax == 0
```

Note: This macro is a convenience that is neither optimized for code size nor speed.

Method scope

Within a method definition, all object members can be called without specifying the object type. The method's object type is assumed by default.

Private methods are only allowed to be called from within method definitions of the same object type or a descendant. Any attempt to call a private method from a different scope will end in a compilation error.

Method calls

The complete syntax to call a method is as follows:

```
OCall InstancePointer::NameSpace:ObjectName.MethodName, Arguments
```

The first parameter is the object instance pointer, the second, separated by a double colon, is the object namespace followed by a single colon and the object name. Finally, the method name, separated by a dot followed by the method arguments.

Depending on the context, some parts of this syntax can be omitted. If the invocation is done within a method implementation, if the instance pointer is omitted, the hidden *pSelf* POINTER is taken. In addition, if the object namespace is omitted, the current is assumed. Equally for the object name.

This flexible syntax save typing time and enhances the readability of the code.

In some cases, when you have overwritten a dynamic method, you would call the original method. This can be achieved using a call to the object's template where the original method address is still stored:

```
TCall esi::Triangle.GetArea
```

On static methods, a *TCall* has no effect, since an *OCall* executes a *TCall* for this type of methods.

In other situations, you will need to call the method of the direct ancestor. This is frequently happens in constructor and destructor methods. You can accomplish this easily by using the *ACall* macro:

```
ACall esi::Triangle.GetArea
```

In this example, this line calls the Shape's *GetArea* method.

Internally *ACall* performs a *TCall* on the ancestor's template. In this way, if the ancestor's method is a dynamic method that was overridden, *ACall* uses the original method address stored in the template!

The last possible way to call a method is directly, circumventing the information stored in its virtual method table. It is the fastest way to invoke a method, but you lose the OOP polymorphism ability. A direct call follows the same syntax as the previous method calls but using the *DCall* macro.

All method calls in x86 mode use the STDCALL calling convention with automatic prologue and epilogue generation. In x64 mode, the FASTCALL calling convention is used.

ObjAsm automatically adjust the stack for variable parameter count (*vararg*).

Like other **ObjAsm** macros, the method calls can be preceded by "$" to return the rax or eax register, used to return a value. The register width depends on the target bitness.

Static or local objects can be called using their symbol in place of the pointer to them. **ObjAsm** computes internally the pointer in the most efficient way to perform the method call.

```
local Object1:$Obj(MyObject)

New Object1
OCall Object1::MyObject.Init, 120
...
```

Method redefinition

All methods types previously defined can be redefined. A method redefinition means that an object descendant modifies the behaviour of a method by rewriting it. Even more, it is possible to change the method arguments. This is especially useful for constructors, but you must be aware, that methods that have different number of arguments cannot be called polymorphically.

A special case is the invalidation of a method. This is done using the *ObsoleMethod* macro. It assigns the value 0FFFFFFFFh to the method address placeholder (VMT / IMT / Instance). If the method has an associated *Event* that triggers the method, this event was also removed from the Event Table.

Abstract methods

Sometimes it is necessary to reserve a method at an ancestor level that are implemented later in its descendants. This strategy makes it possible to call this method for all descendant in the same way (polymorphism), independently of other changes in the structure of the derived objects.

The same applies to common variables. If you plan to use polymorphism, try to place all common variables in the common ancestor object.

The address of an abstract method is set to 0FFFFFFFFh. Any try to call it will end in an exception.

Syntax simplification

In some cases, depending on the current scope, it is possible to omit some parameters of the method call.

If the method call is done within a method implementation and the instance pointer is omitted, the *pSelf* pointer is assumed.

In this same context, if the object name (second parameter) is omitted, the current object is assumed.

```
Method ObjectX.Get, uses esi, dIndex:dword
  SetObject esi
  OCall CheckIndex, dIndex
  …
```

In this example, the OCall is interpreted as:

```
OCall pSelf::ObjectX.CheckIndex, dIndex
```

A little faster alternative syntax, but with the same result would be:

```
OCall xsi.CheckIndex, dIndex
```

Method prologue and epilogue

In 32-bit mode, methods usually use the standard prologue and epilogue supplied by MASM®, but when the TRACE option is activated using the *SysSetup* macro, the prologue and epilogue are modified to perform method tracing and performance monitoring. Be aware of this when you supply your own prologue or epilogue code.

Methods in 32-bit mode also support an option called NOFRAME that disables the prologue and epilogue generation. Normally this is equivalent to *prologue:none* and *epilogue:none* MASM® directives, but if the *TRACE* option is activated, only the tracing code is inserted. It is the coder responsibility to adjust the stack and to access the method parameters the right way!

Path customization

The "Code" folder contains a file called OA_SetupXX.inc that defines all standard paths used by the **ObjAsm** programming model.

Conventions

To maintain certain degree of consistency over the whole model, some conventions has been taken:

- Objects were written in .inc files.
- Compilation files were written in .asm files.
- Object IDs are located together in the ObjIDs.inc file. Custom IDs should be maintained in a separate file like MyObjIDs.inc.
- Objects error codes are located together in the ObjErrs.inc. Custom code should be maintained in a separate file.

All objects that support steaming capabilities (derived from *Streamable*) MUST have an ID.

8. Bitness neutral code

ObjAsm provides a code library called **ObjMem** in two flavours, 32-bit and 64-bit each of them in the corresponding “Code/ObjMem/32” and “Code/ObjMem/64” directory. A third directory called “Code/ObjMem/X” contains code that can be directly assembled in the target bitness. This is possible due to the use of a set of equates that maps the some type and register names to the correct word size. E.g. the xax register name is mapped to eax in 32-bit mode, whereas to rax in 64-bit mode. To simplify the use of some variables, the XWORD type was introduced. It is equivalent to the *size_t* C++ type.

The vast majority of the supplied objects, demos and projects were coded using BNC.

This feature is supported by the translated Windows® header files which was created using macros that change the API definitions accordingly to the target bitness.

9. Backstage work

The last line in an object definition is the *ObjectEnd* macro, the most important macro of **ObjAsm**. This is where the real OO work begins.

For each defined object, a metadata structure is created in the data segment that contains the entire object data. This structure contains the object template, the DMT, IDs and dispatch data. The first element in this structure is a pointer to the next object metadata, building a chain that starts with the pointer called p1stOMD. The last object metadata holds a NULL in this place indicating that it is the last element in the chain.

The chain can be extended with loaded objects from a DLL.

The returned value in the rax/eax register is the pointer to this memory block containing the object instance.

The first data of the object instance is a pointer to the Dual Method Table. This table consists of two different structures.

The first, with positive offsets from the DMT pointer, is the Interface Method Table (IMT), which is compatible with the COM specification.

The second structure, with negative offsets, is the Virtual Method Table (VMT) used to store the remaining virtual method addresses.

This construct allows the simultaneous use of both tables. Inheritance and polymorphic properties are guaranteed independently of both method tables.

10. Object placement and lifecycle

Object instances are memory constructs that can be located in different places. Usually they are instantiated as standalone entities and their lifecycle are determined by the “New” and “Destroy” macro invocations.

Less obvious are object instances created by resources, better said as part of i.e. a resource dialog. In this case, **ObjAsm** limit their lifecycle between the WM_NCCREATE and WM_NCDESTROY messages sent to those objects.

Objects instances can also be created on stack memory, like variables within a method or procedure that are created with the “local” keyword. The necessary stack space to hold the object instance is reserved automatically by the prologue code at virtually no CPU cost. Since the instance memory is freed by the epilogue code when the method or procedure finished , it is not necessary to call the *Destroy* macro.

The prologue code simply reserves the necessary amount of stack memory, but does not initialize it. This is the responsibility of the *New* macro when it is called in this context. Usually it follows the constructor method to initialize additional resources. Before the instance memory is released, the destructor must be called to free the previously allocated resources!

```
local Object1:MyObject

New Object1::MyObject
OCall Object1::MyObject.Init, 120
...
```

An object can be nested with the *Embed* macro into another object. This case is different from a local object because the nested object stays alive with its host that controls its initialization and finalization. A nested object should never be freed with the *Destroy* macro.

```
Object MyApp,, Primer
...
Embed SBar, StatusBar
...
ObjectEnd
```

For a more flexible use of the method invoking, the *OCall* macro can accept as first parameter an instance pointer or a symbol of the local or nested object. In this last case, the instance pointer is calculated in the most efficient way.

```
OCall MyApp.SBar::StatusBar.Init
```

11. Method binding

ObjAsm supports "*Early Binding"* and "*Late Binding"*. These terms, used widely in the OOP jargon, refer on when and how the method execution addresses are resolved.

"Early Binding" refers to methodology in which the addresses are resolved on compile-time. The method invocation is directly done using the shortest and fastest way.

"Late Binding" refers to the methodology in which the addresses are resolved at run-time using tables called "Virtual Method Tables". First, the execution address is read from the table and then the invocation is done in the same way as using early bound methods. Late bound methods are more flexible since they can be manipulated over the Method tables but at a small cost that is the few instructions needed to read those tables.

Virtual, *Dynamic* and *Interface* methods are late bounded. Only Static methods are early bounded. That means that the binding type is selected when the object layout is designed.

The binding type does not affect inheritance and parenthood relations, but special care has to be taken since some polymorphic attributes are not preserved. Only late bound methods can be considered for polymorphic purposes.

12. Dispatching mechanism

When the OS sends a message, it is processed by the main window procedure that is registered to the OS when a window class is created.

For all applications that derive from *SdiApp*, *MdiApp* or *DlgApp*, this mechanism is completely implemented in the *Run* and *WndProc* methods.

The former is the message pump while the second is the dispatcher.

Using a derived from the above mentioned objects has some advantages over regular window procedures since it implements an event method table (EMT) that is used to dispatch the OS messages directly to the handler method. The link between the message ID (e.g. WM_PAINT) and of the handler method is done using the following syntax:

```
Event  OnClick,  WM_CLICK
```

This macro links the procedure *OnClick* with the event ID *WM_CLICK*.

It is also possible to link a method to more than one event ID, e.g.:

```
Event  OnClick,  WM_CLICK, WM_DBLCLK
```

When the WndProc method recognises one of these events, the corresponding method is called passing then event ID in the eax register.

Derived objects also inherit the event translation table (ETT) the same way that methods and variables do.

If a handler method is redefined, the new code is called instead of the old method.

All event handler methods must have the same prototype:

```
Method App.OnClick,, wParam:dword, lParam:dword
  …
  …
MethodEnd
```

The dispatching mechanism looks like the following code:

```
Method XXX.WndProc, uses esi, uMsg:dword, wParam:dword, lParam:dword
  .if uMsg == WM_NCCREATE
    mov eax, lParam
    mov esi, [eax].CREATESTRUCT.lpCreateParams
    or esi, esi
```

```
      jz @@DefProc
      m2m [esi].SdiApp.hWnd, pSelf
      invoke SetWindowLongPtr, pSelf, GWL_USERDATA, esi
    .else
      invoke GetWindowLong, pSelf, GWL_USERDATA
      .if eax == 0
        invoke GetWindowLongPtr, pSelf, DWL_USER
        or eax, eax
        jz @@DefProc
      .endif
      mov esi, eax
    .endif
    DispatchEvent <DefWindowProc, pSelf>
  MethodEnd
```

The first thing the procedure does is to check if the received event ID is *WM_NCCREATE*.

If it is the case, the object instance pointer, which is passed through the CREATESTRUCT structure, is stored in the Window instance USERDATA. In some cases, this is not possible and a Property attached to the Window instance is used to store this information.

If the message is not *WM_NCCREATE*, the procedure will try to retrieve the object instance pointer. If this is not successful, the default window procedure *DefWindowProc* is called.

Next, the ETT is searched for the received event ID and the address of the corresponding method is retrieved. This method is called, passing the instance pointer (*pSelf*), *wParam*, and *lParam* as parameters.

The method return value is used as the *WndProc* return value.

Example

DemoApp02 is a descendant of the SDI application object (SdiApp), which is used as base object for this example. DemoApp02 defines three events with their respective methods:

```
  VirtualMethod  OnCommand, LParam, WParam
  VirtualMethod  OnClose,   LParam, WParam
  VirtualMethod  OnPaint,   LParam, WParam

  Event OnCommand, WM_COMMAND
  Event OnClose,   WM_CLOSE, WM_QUERYENDSESSION
  Event OnPaint,   WM_PAINT
```

The first event method (*OnCommand*) is triggered each time the *WndProc* receives a *WM_COMMAND* message, which can be generated in many ways, i.e. when a menu item is clicked: in this demo, the Exit and About menu items are caught in the *OnCommand* method.

The second event method (*OnClose*) may be triggered by two messages, *WM_CLOSE* or *WM_QUERYENDSESSION*. The method pops up a messagebox asking the user to close the application.

The third event method (*OnPaint*) is triggered by the *WM_PAINT* message when the application needs to be redrawn. The method draws the background and an inner edge and fills the client area of the application window.

13. OOP macros and procedures

Object

The reserved word *Object* is used to define objects that conform the **ObjAsm** model. An object is a structure that contains a fixed number of members. Each member is either a variable, which contains data of a particular type, or a method, which performs an operation on these data.

An object can inherit members from another object. The inheriting object is a descendant and the object inherited from, is an ancestor.

ObjectEnd

The reserved word *ObjectEnd* is used to terminate the object definition started with Object.

VirtualMethod

The reserved word *VirtualMethod* is used to define a method whose entry point is resolved at run-time (late binding), using the address stored in the object Virtual Method Table (VMT). A static method cannot be redefined at run-time at an object's instance level.

VirtualAbstract

The reserved word *VirtualAbstract* is used to define a static method whose implementation is not defined in the object declaration in which it appears; its definition is instead deferred to descendant objects. An abstract method defines only placeholder for an execution address in the VMT, but not the underlying method code.

Overriding an abstract method is identical to override a normal virtual or dynamic method, except that in the implementation of the method, an inherited method is not available to be called (see *ACall*).

Calling an abstract method through an object that has not overridden the method will generate an exception at run time, since the execution address is set to FFFFFFFFh. You can provide your own address to handle such situations, i.e. displaying an error message.

DynamicMethod

The reserved word *DynamicMethod* is used to define a method whose entry point is resolved at run-time using the address stored in the object's instance. This makes possible to redefine this address at run-time for each instance.

DynamicAbstract

Same as *VirtualAbstract*, but applied to a dynamic method.

InterfaceMethod

The reserved word *InterfaceMethod* is used to define a special method to invoke COM compliant methods.

RedefineMethod

The reserved word *RedefineMethod* is used to redefine a method into an object declaration. The inherited code is replaced with the new declared procedure.

StaticMethod

The reserved word *StaticMethod* is used to define a method whose entry point is resolved at compile-time. The concept of bound methods is equivalent to "early binding" in other languages. The advantage of these methods is that the execution time is slightly shorter.

InlineMethod

The reserved word *InlineMethod* is used to define a method that works like a macro. An inline method acts like a macro. It expands where the call is made. Like static methods, no polymorphism can be applied. The method definition must be done similar to a macro.

ObsoleteMethod

The reserved word *ObsoleteMethod* is used to strip a method from an object declaration.

DefineVariable

The reserved word *DefineVariable* is used to reserve data storage for an object member of any type. This variable can be initialized at compile time to reduce the constructor code size and execution time.

Embed

The reserved word *Embed* is used to reserve data storage for an object inside another object like another variable defined with *DefineVariable*. The embedded object members are automatically initialized at compile time.

Event

The reserved word *Event* is used with Windows applications to trigger some method, when a corresponding message is sent to the WndProc procedure.

A shorter way to define an Event and a corresponding method is using the *VirtualEvent* or *DynamicEvent* for a *VirtualMethod* or a *DynamicMethod* respectively.

Override

The reserved word *Override* is used to redefine a static or dynamic method at run-time.

Redefining a static method will affect all instances of the object, while redefining a dynamic method only affects a specific object instance.

($)ICall

The reserved word *ICall* is used to call an interface method following the rule of COM method invocation.

($)OCall

The reserved word *OCall* is the normal way to call a method. If it is a static or private method, the execution address is taken from the object template, while if it is a dynamic method, the execution address is taken from the object instance data.

($)TCall

The reserved word *TCall* is used to call a method using always the execution address stored in the object template, regardless of the value stored in the object instance data.

($)ACall

The reserved word *ACall* is used to call an ancestor method using the execution address stored in the ancestors object template.

($)DCall

The reserved word *DCall* is used to call a method, linking it directly at compile-time. No object member is used to call the procedure.

($)MethodAddr

The reserved word *MethodAddr* is used to obtain the execution address of a method.

($)New

The reserved word *New* is used to create an instance of an object. A constructor can be supplied to be executed immediately after the instantiation.

Destroy

The reserved word *Destroy* is used to safely free an allocated object instance. Before the memory is released, the destructor *Done* is called.

Kill

The reserved word *Kill* is used to free an allocated object instance. No validation check is performed. Before the memory is released, the destructor *Done* is called.

SetObject

The reserved word *SetObject* is used to assign an object type to a register.

ReleaseObject

The reserved word *ReleaseObject* is used to release a previous assignment done with *SetObject*.

GetObjectID

This is a procedure to retrieve the object's ID, specified with the *Object* macro.

SysInit

This is a macro that must be placed at the beginning of every application. It is the runtime initialization of the OOP model. First, it loads the *hInstance* and *hProcessHeap* variables and starts the object chain setting the *pFirstObject* pointer.

Finally, it invokes all defined *Startup* methods. When this special BoundMethod is called, it passes a pointer to the object template as instance pointer (*pSelf*). That allows performing some special initialization on the template or some registration work to the OS.

SysDone

This macro is the counterpart of *SysInit* and must be placed at the end of every application. It invokes the declared Shutdown methods of all objects. The instance pointer (*pSelf*) points again to the object template. This allows the application to perform some work on resources or some unregistration work to the OS.

SysSetup

This macro is required to setup the OOP model. The first argument defines the desired OOP level, loading the corresponding base objects and macros. The second argument can be the DEBUG macro, which indicates that the debugging subsystem should be loaded. This macro has also some arguments to customize this system and what debugging subsystems to use.

Method

The reserved word *Method* is used to indicate that the following lines of code are part of an object method. It also sets the scope, necessary for other macros. The syntax is similar to the *proc* syntax.

MethodEnd

The reserved word *MethodEnd* is used to indicate the end of a method. It finalizes all assumptions on registers which were done with previous *SetObject* macros, sets the End Of Method (*@@EOM*) label and executes a *ret* instruction. An optional parameter can modify the ret instruction.

14. Model features

The **ObjAsm** programming model supports three fundamental build-in features:

- Error propagation.
- Data and object streaming.
- Data or object conglomeration with collections.

Error propagation

All objects have a variable inherited from Primer called pOwner that can point to an object that is responsible for it. This data member is usually set during initialization and does not change during the life cycle of the object.

To implement the error propagation, 2 additional data members were added to Primer object:

```
DefineVariable   dErrorCode,   DWORD,    0
DefineVariable   pErrorCaller, POINTER,  NULL
```

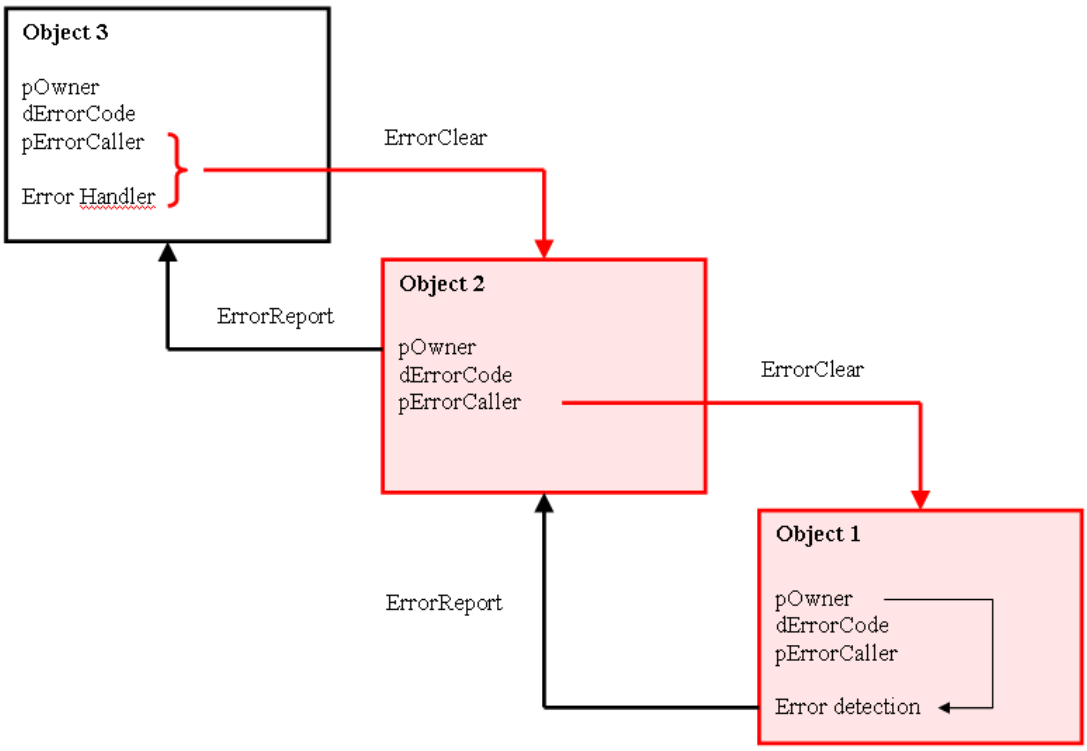
and additionally 3 methods to handle them

```
VirtualMethod    ErrorClear
VirtualMethod    ErrorReport,   POINTER, DWORD
VirtualMethod    ErrorSet,      DWORD
```

Supposing that a method has detected an error condition, it sets the error code using the *ErrorReport* method and advises to its owner that something has gone wrong. In this situation, the owner can react taking a corrective action or it can pass the error signalization to its own owner. This behaviour is repeated until some object can handle the error situation or the top most owner was reached.

The *ErrorSet* method was used by the *ErrorReport* method to simply set the error code without propagation.

The *ErrorClear* method was used to walk the error path back to the first caller, resetting the error code.



Data and object streaming, persistence and serialization

Streaming is a widely known technique used to store data into streams. This also known as persistence.

A stream is a generic name of a storage medium, like a disk file, a memory block or even a communication port.

The goal of using a generic object for all types of possible storages is to handle all of them in the same way. Generic code to load and store the object data has the big advantage that the code of the storage stream can be made later or can be redirected to a different stream type.

To get streaming capabilities, an object has to derive from *Streamable*. This object defines two abstract methods, *Load* and *Store*.

These methods must be redefined to provide data persistence. These methods are automatically called when the stream invokes the Put or Get methods to load or store the object data on the stream.

ObjAsm defines several *Stream* objects like *DiskStream*, *MemoryStream*, etc. with different specializations. The use of each stream type depends on the specific purpose.

Persistence is implemented in the **ObjAsm** model using serialization. This technique enables to write the data sequentially and read them back in the same sequence. For most data types, this is trivial, but for pointers and references, which may be loaded in a different process, they are completely meaningless. **ObjAsm** provides a special mechanism to restore these elements in a proper way to get back their original functionality.

Data or object conglomeration using collections

One of the most repeated situation is the requirement to bundle data or even objects together. For simple data structures, arrays appear to be the best solution, but for complex structures with different sizes, or when the array dimension size cannot be determined a priory, a better solution must be found. *Collections* are the answer!

Collections are flexible and highly optimized objects that can handle all types of structures and objects at runtime. If necessary, you can increase your storage capacity up to a predetermined limit. Automatic object destruction is also provided when the collection members are no longer needed.

ObjAsm offers different types of *collections* that specialize in different tasks. The basic form of a *collection* can handle objects as described above. A *SortedCollection* handles the items applying a sorting strategy. A *StrCollection* is a specialized derived object that handles strings. Finally, a *DataCollection* specializes in data structures that do not require special handling, such as objects.

All types of *collections* can be customized at runtime overwriting the dynamic methods *DestroyItem*, *GetItem* and *PutItem*.

15. Debugging

ObjAsm provides a macro based debug system that can output information to some devices. Depending on the configuration, the output can be directed to a window (DebugCenter), a console window or to a file. This is activated using the DEBUG keyword in the SysSetup macro indicating in brackets WND, CON or LOG respectively.

```
SysSetup OOP, WIN64, WIDE_STRING, DEBUG(WND)
```

Additional keywords can be specified to activate special features like:

- INFO: displays additional information
- TRACE: activates method tracing (required for DbgTraceObject and DbgTraceShow)
- RESGUARD: activates the API monitoring (32-bit).
- STKGUARD: activates the stack protection system (32-bit).
- TIMESTAMP: displays time information.
- "name": specifies the DebugCenter child window name.

Macros

All macros directs the output to the “DestWnd” window.

Strings

- DbgLine "DestWnd": displays a separation line.
- DbgText, "Text", "DestWnd": displays a given text.
- DbgWarning "Text", "DestWnd": displays a given text using the red color.
- DbgStr pStr, "DestWnd": displays a string pointed to by the first argument.

Integer numbers

These macros display variables (register or symbols, max. 64 bits) followed by an additional information string.

- DbgBin Variable, "Info", "DestWnd": displays the content of variable in binary format.
- DbgDec Variable, "Info", "DestWnd": displays the content of variable in decimal format.
- DbgHex Variable, "Info", "DestWnd": displays the content of variable in hexadecimal format.

Floating point numbers

This macro displays the content of real4 and real8 variables.

- DbgFloat Variable, "Info", "DestWnd": displays the content of variable in scientific notation.

GUID numbers

- DbgGUID pGUID, "Info", "DestWnd": displays the GUID number pointed to by pGUID.

Bitmaps

- DbgBmp hBmp, "DestWnd": displays the bitmap referred to by hBmp (only possible using DebugCenter).

Memory

- DbgMem pMem, Size, format, "Info", "DestWnd": displays the content of the memory block pointed to by pMem.

  The output format can be specified with one of the following constants:

  1. DBG_MEM_STR,
  2. DBG_MEM_UI8, DBG_MEM_UI16, DBG_MEM_UI32,
  3. DBG_MEM_I8, DBG_MEM_I16, DBG_MEM_I32,
  4. DBG_MEM_R4, DBG_MEM_R8

- DbgGlobalMemUsage "Info", "DestWnd": displays the status of global memory.

Hardware

- DbgFPU "Info", "DestWnd": displays the complete status of the FPU.

API

- DbgApiError "Info", "DestWnd": displays API error status that can be retrieved using the GetLastError API.
- DbgComError dError, "Info", "DestWnd": displays COM error based on the dError value.

Object

- DbgObject Instance::Class, "Info", "DestWnd": displays the content of all members of the given object instance.
- DbgTraceObject Instance / DbgTraceShow Class, "Info", "DestWnd": displays how many times the methods of a specific object instance were called and the number of CPU cycles it took to execute. Tracing is limited to only one object instance!

Helpers

- Fix "Text": echoes a message to the compilation output device.
- ASSERT Argument, "Info", "DestWnd": Displays a message if the argument value is FALSE.
- DbgClearAll: Clears the output device.

16. Tool chain

ObjAsm has build tools to facilitate the assembly process. In the „…\ObjAsm\Build" directory are all required .cmd files to compile resources, xml files, .asm files, build libraries, etc.

17. Manual installation

Follow these instructions:

1. Get Visual Studio Community from the following link: https://www.visualstudio.com/. Install it to get an x64-debugger and the latest tool chain from Microsoft®.
2. Get Windows Kits SDK from the following link: https://developer.microsoft.com/de-de/windows/downloads/windows-10-sdk and install it to get additional tools.
3. Get the ObjAsm package from http://objasm.x10host.com and install it. Depending on the destination drive and subdirectory, some paths may have to be adjusted.
4. Get UASM64.exe and UASM32.exe from http://www.terraspace.co.uk/uasm.html and install it in the "...\ObjAsm\Build\Tools" folder.
5. Copy the necessary .lib files from the sdk (2) in the "...\ObjAsm\Code\Lib\32\Windows" and "...\ObjAsm\Code\Lib\64\Windows" folders.
6. Create the following environment variables and adjust the paths to your installation:
   a. OBJASM_PATH = C:\ObjAsm
   b. MSVS_PATH = C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin
   c. WINKIT_PATH = C:\Program Files (x86)\Windows Kits\10\bin\10.0.16299.0
   d. Reboot if necessary.
7. Adjust the paths in the file "...\ObjAsm\Build\OA_SET.cmd" accordingly to the previous installation.
8. Compile the ObjMem library using the file „...\ObjAsm\Code\ObjMem\MakeObjMem.cmd"
9. Optionally compile the objects using the file „...\ObjAsm\Code\Objects\MakeObjects.cmd"
10. Optionally compile the Demo-Applications and Projects using the Make.cmd file in each directory

Notes:

1. Some applications may have hard coded paths. Adjust them before compiling.
2. Some .inc files cannot be used as they are. In most cases, only the API-Name-Prefix must be changed.

18. Component Object Model (COM)

Introduction

The component object model (COM) is an API specification introduced by Microsoft®, which provides a standard way for interfacing component objects and their programming language-independent implementation.

The component object model API is provided by a DLL called COMPOBJ.DLL. When you use this API to create an instance of a component object, you receive a pointer to a method table, also known as interface method table (IMT). You can use these methods as if they were methods of any object type.

A component object can be implemented by an EXE or DLL, although the implementation is transparent to you, because of an isolation process, provided by OLE 2, called marshaling. OLE 2's marshaling mechanism handles all the intricacies of calling methods across process boundaries, so it is safe for example to use a 64-bit component object from a 32-bit application.

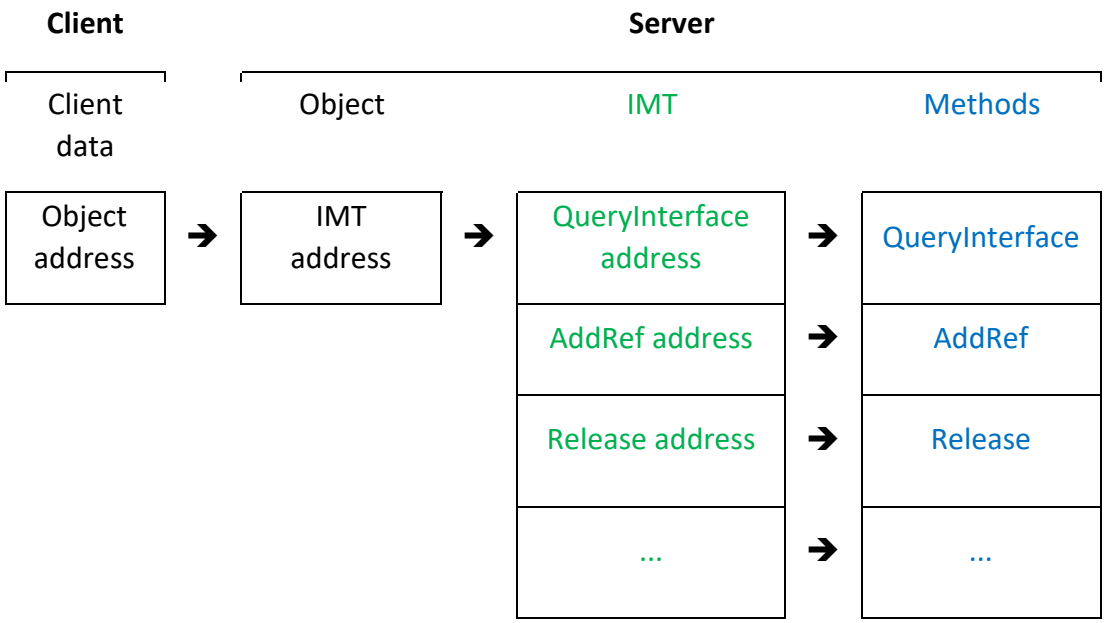
Interfaces

COM defines how OLE object methods are mapped in memory. Method pointers are arranged in tables. The description of a table in any programming language is called an interface.

ObjAsm introduces a DMT that is fully compatible with the COM specification.

When you create an object instance, the first member in the allocated memory, is a pointer to the IMT.

This way, it is possible to call a COM method following the standard established in the specification:



ObjAsm introduces a new invocation macro called *ICall*, which calls a COM method following the specification illustrated above. ICall uses solely the structure definitions provided by the Windows® APIs or the object interface definition. The interface name is not mangled. The members are!

Defining a COM interface of an object unrelated interface can be done using the *BEGIN_INTERFACE* / *END_INTERFACE* macros. No more information is required!

The following example shows two of the primer COM interface layouts.

```
BEGIN_INTERFACE IUnknown,, <00000000-0000-0000-C000-000000000046>
  STDMETHOD    QueryInterface,       POINTER, POINTER
  STDMETHOD    AddRef
  STDMETHOD    Release
END_INTERFACE

BEGIN_INTERFACE IErrorInfo, IUnknown, <1CF2B120-547D-101B-8E65-08002B2BD119>
  STDMETHOD    GetGUID,              POINTER
  STDMETHOD    GetSource,            POINTER
  STDMETHOD    GetDescription,       POINTER
  STDMETHOD    GetHelpFile,          POINTER
  STDMETHOD    GetHelpContext,       POINTER
END_INTERFACE
```

The first argument of the interface header is the name, followed by an optional parent interface from which it inherits the first methods. This is the case of the second layout, which inherits all methods from the first and adds 2 more. The last optional argument is the associated IID, which is defined as a symbol like sIID_InterfaceName. In this way, the IID, if needed more than once, can be easily stored in the .const or .data segment using the following macros:

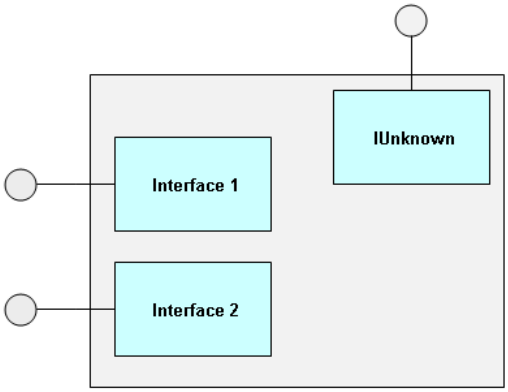
```
.const
DefGUID IID_IUnknown, %sIID_IUnknown
DefGUID IID_IErrorInfo, %sIID_IErrorInfo
…
```

Note: other GUIDs, like Class IDs (CLSID) can be defined in the same way.

Component

ObjAsm uses objects to implement any interface that belongs to a component. Since many of them are used in more than one component, it makes sense to write them in a form that can be reused with little customization.

A graphical representation of the **ObjAsm** COM model looks like:



The boundary represents the component object. The circles are the interfaces that are implemented by the interface objects that are inside the component object. All interface objects are grouped together in a collection.

Each of these interface objects has a unique IID that is used to identify them. The component object is identified by its CLSID.

Note that a component never has less than one interface, which must always be present. This interface is IUnknown.

Implementations

Interface implementation

Each interface should derive directly or indirectly from the IUnknown interface object. The variables that are necessary for the functionality of the interface can be stored in the object itself or in the component object. The component object can be reached using the Owner pointer or the Owner's Owner pointer, depending on the implementation.

The implementation of the IUnknown interface object holds the basic three interface methods: QueryInterface, AddRef and Release and 2 variables: a pointer to a GUID and the reference count. Each interface is responsible to hold its own reference count!

Two additional static methods are added to provide the full functionality of the component: CanNotRelease and GetInterface. Both are iterator methods and are called from a FirstThat collection method. Do not call them directly!

Component implementation

To guarantee a full functionality, each component object has to inherit directly or indirectly from the Component object.

It contains three variables: a pointer to a GUID that identifies the component class, a collection to append all supported interfaces, and a pointer to an external IUnknown interface for aggregation purposes.

The component object has two new methods besides Init and Done. There is *CanNotDestroyNow*, which iterates through all interfaces to check if all reference counts are

zero, and *GetInterface* to check if a particular IID is supported by that component. This last method is used by the *QueryInterface* interface method.

In Process Server implementation

A component object can be implemented as an In Process Server. This means a DLL that can be used by all applications that follow the COM standard.

All the code required to register and unregister the server, the instantiation of the Class Factory, and the internal DLL reference Count are grouped together in a file called COM_DLL.inc, which is required to create an In Process Server. Another required file that is responsible for assembling the DLL is the definition file COM_DLL.def. This file declares all public procedures contained in the DLL.

The component (server) retrieval mechanism uses another component called "Class Factory" provided by the DLL. It is responsible for the instantiation of the requested components.

Class Factory implementation

Like all other components, the Class Factory component has at least two interfaces, *IUnknown* and *IClassFactory*.

Registration

A DLL requires two registrations.

Since more than a server can be contained in a single DLL, a registration-chained structure is used when the *RegisterServerInfo* macro is invoked. This macro can be found in the **COM.inc** file. Its arguments are:

1. The object name of the component that will be instantiated
2. Its type library GUID
3. Server major version
4. Server minor version
5. Server Prog ID string
6. Server description
7. Verb Map pointer

The second registration to perform is the registration of the required type libraries. This is done with the RegisterTypeLibrary macro, which is also included in the COM.inc file. Its arguments are:

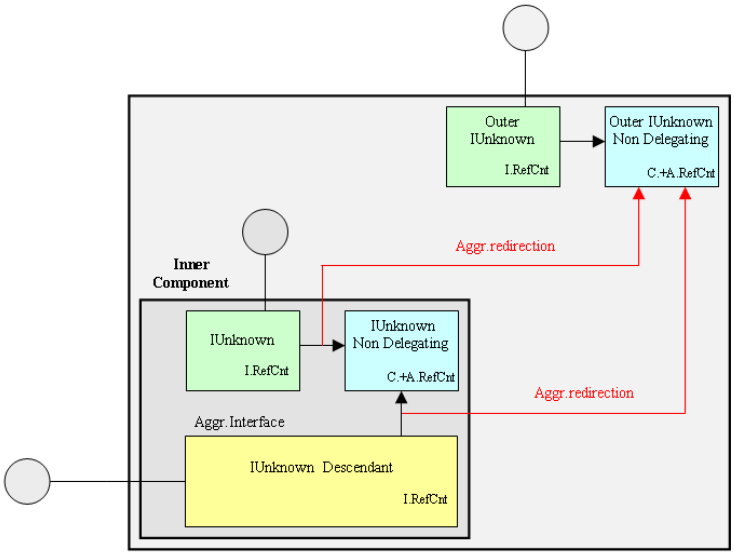
1. Resource ID
2. Type Library GUID
3. Major version
4. Minor version

Aggregation

Aggregation in the COM context is a strategy that presents an interface to a foreign component as if it were the own. To accomplish this goal, several tricks must be performed to convince the component client that it is talking to a single component. Special care should be

taken with the *IUnknown* interface that was divided into two interfaces, the *delegating* and the *non-delegating IUnknown* interface.

The strategy proposed by the COM designers is to redirect the information flow from the inner component *non-delegating IUnkown* to the outer *non-delegating IUnkown*. By this way, proper interface counting can be guaranteed and the *IUnknown* methods of the inner component are replaced by those of the outer component. The client never sees the implementation of the *IUnknown* methods of the inner component.



19. Recommendations

The following is a list of recommendations when programming using the **ObjAsm** to prevent conflicts with future releases:

Object IDs

User object IDs has to be greater than 80000000h. The range from 0 to 80000000h is reserved for **ObjAsm** objects.

Object Error constants

The object errors are constants that can be found in the ObjErrs.inc file. Following Microsoft recommendation about error codes, bit 29 of those values must be set to signal a user defined value.

If you add more errors constants for your own objects, avoid repetitions using the range above 30000000h. The range from 20000000h to 30000000h is reserved for internal use.

Files

Do not modify ObjIDs.inc nor ObjErrs.inc files. Create your own files to add more functionality since these files will be overwritten by future updates.