LuaCOM User Manual

(Version 1.0 RC1)

Vinicius Almendra Renato Cerqueira

4th June 2003

Contents

Chapter 1

Introduction

LuaCOM is an add-on library to the Lua language that allows Lua programs to use and implement objects that follow Microsoft's *Component Object Model* (COM) specification **and** use the *ActiveX technology* (or OLE automation) for property access and method calls.

1.1 Features

Currently, the LuaCOM library supports the following features:

- dynamic instantiation of COM objects registered in the system registry, via the `luacom_CreateObject` function;
- dynamic access to running COM objects via `luacom_GetObject`;
- COM method calls as normal Lua function calls;
- property access as normal table field access;
- type conversion between OLE Automation types and Lua types;
- object disposal using Lua garbage collection mechanism;
- implementation of COM interfaces and objects using Lua tables;
- use of COM connection point mechanism for bidirectional communication and event handling;
- fully compatible with Lua 4;
- use of COM objects without type information.

1.2 How to use

Using LuaCOM is straightforward: you just have to link your program with LuaCOM's library, include the LuaCOM's header — `luacom.h` — and call the proper initialization and termination functions before using any of LuaCOM's functionalities. Here is an example of a simple `C` program using LuaCOM.

```
/*
 * Sample C program using luacom
 */
#include <stdio.h>
#include <ole2.h> // needed for OleInitialize and OleUninitialize
#include <lua.h>

#include "luacom.h"

int main (int argc, char *argv[]) {

  /* COM initialization */
  CoInitialize(NULL);

  /* library initialization */

  lua_State *L = lua_open(0);

  luacom_open(L);

  if(lua_dofile("activex_sample.lua") != 0) {
    puts("Error running sample.lua!");
    exit(1);
  }
  luacom_close(L);
  lua_close(L);

  CoUninitialize(NULL);
  return 0;
}
```

Notice that it's necessary to initialize COM before `luacom_open` and to terminate it only after the last `lua_close`, otherwise faults may occur.

Chapter 2

LuaCOM Elements

LuaCOM is composed by the following elements:

- LuaCOM API, used primarily to initialize the library, create objects, implement ActiveX interfaces in Lua and to manipulate connection points;
- LuaCOM objects, which make available in Lua ActiveX objects and interfaces;
- ActiveX binding, which translates accesses on LuaCOM objects to ActiveX interface calls and ActiveX accesses on an interface implemented in Lua to Lua function calls or table accesses;
- LuaCOM type conversion rules, which govern the type conversion between Lua and ActiveX values;
- LuaCOM parameter passing rules, which describe how LuaCOM translate a Lua parameter list to a COM one and vice versa.

2.1 LuaCOM API

The LuaCOM API is divided in two parts: the Lua API and the C/C++ API. The C/C++ API is used primarily for initialization of the library and for low-level construction of LuaCOM objects. The Lua API permits Lua programs to access all the functionality of LuaCOM. Below there is summary of the LuaCOM API. Detailed information on these functions is available in chapter 5.

Lua API

Function	Description
`luacom_CreateObject`	Creates a LuaCOM object.
`luacom_NewObject`	Creates a LuaCOM object implemented in Lua.
`luacom_GetObject`	Creates a LuaCOM object associated with an instance of an already running ActiveX object.
`luacom_ExposeObject`	Exposes a LuaCOM object, so that other applications can get a reference to it.
`luacom_RevokeObject`	Undoes the operation of `luacom_ExposeObject`.
`luacom_RegisterObject`	Fills in the registry entries necessary for exposing a COM object.
`luacom_Connect`	Creates a connection point between an object and a Lua table.
`luacom_ImplInterface`	Implements an IDispatch interface using a Lua table.
`luacom_ImplInterfaceFromTypelib`	Implements an IDispatch interface described in a Type Library using a Lua table.
`luacom_addConnection`	Connects two LuaCOM objects.
`luacom_releaseConnection`	Disconnects a LuaCOM object from its connection point.
`luacom_isMember`	Checks whether a name correspond to a method or a property of an LuaCOM object.
`luacom_ProgIDfromCLSID`	Gets the ProgID associated with a CLSID.
`luacom_CLSIDfromProgID`	Gets the CLSID associated with a ProgID.
`luacom_GetIUnknown`	Returns an `IUnknown` interface to a LuaCOM object as a userdata.
`luacom_DumpTypeInfo`	Dumps to the console the type information of the specified LuaCOM object. This function should be used only for debugging purposes.

C/C++ API

Function	Description
`luacom_open`	Initializes the LuaCOM library in a Lua state. It must be called before any use of LuaCOM features.
`luacom_close`	LuaCOM's termination function.
`luacom_detectAutomation`	This function is a helper to create COM servers. It looks in the command line for the switches "/Automation" and "/Register" and call some user-defined Lua functions accordingly.
`luacom_IDispatch2LuaCOM`	Takes an IDispatch interface and creates a LuaCOM object to expose it, pushing the object on the Lua stack.

2.2 LuaCOM objects

LuaCOM deals with *LuaCOM objects*, which are no more than a Lua table with the LuaCOM tag and a reference to the LuaCOM C++ object; this one is, in turn, a proxy for the ActiveX object: it holds an `IDispatch` pointer to the object and translates Lua accesses to ActiveX calls and property accesses. Here is a sample where a LuaCOM object is used:

```
-- Instantiate a Microsoft(R) Calendar Object
calendar = luacom_CreateObject("MSCAL.Calendar")

-- Error check
if calendar == nil then
  print("Error creating object")
  exit(1)
end

-- Method call
calendar:AboutBox()

-- Property Get
current_day = calendar.Day

-- Property Put
calendar.Month = calendar.Month + 1

print(current_day)
```

```
print(calendar.Month)
```

LuaCOM objects can be created using the LuaCOM Lua API; there are a number of functions that return LuaCOM objects. The most relevant ones are `luacom_CreateObject` and `luacom_GetObject`. LuaCOM objects may also be created on demand implicitly, when a return or output value of a COM method is a `dispinterface`.

LuaCOM objects are released through Lua's garbage collection mechanism, so there isn't any explicit API function to destroy them.

A LuaCOM object may be passed as an argument to method calls on other LuaCOM objects, if these methods expect an argument of type `dispinterface`. Here is a sample to illustrate this situation:

```
-- Gets a running instance of Excel
excel = luacom_GetObject("Excel.Application")

-- Gets the set of worksheets
sheets = excel.Worksheets

-- gets the first two sheets
sheet1 = sheets:Item(1)
sheet2 = sheets:Item(2)

-- Exchange them (here we pass the second sheet as a parameter
-- to a method)
sheet1:Move(nil, sheet2)
```

There are two kinds of LuaCOM objects: *typed* and *generic* ones. The typed ones are those whose COM object has type information. The generic ones are those whose COM object does not supply any type information. This distinction is important in some situations.

2.3 ActiveX binding

The ActiveX binding is responsible for translating the table accesses to the LuaCOM object to ActiveX interface calls. Besides that, it also provides a mechanism for implementing ActiveX `dispinterfaces` using ordinary Lua tables.

2.3.1 Implementing `dispinterfaces` in Lua

The ActiveX binding has a `C++` class that implements a generic `IDispatch` interface. The implementation of this class translates the method calls and property accesses done on the objects of this class to Lua calls and table accesses. So, one may implement an ActiveX interface entirely in Lua provided it has a type library describing it. This type library may be a stand-alone one (referenced by its location on the file system) or may be associated with some registered component. In this case, it may be referenced by the `ProgID` of the component.

The `C++` objects of this class can be used in any place where an `IDispatch` or `IUnknown` interface is expected. Follows a sample implementation of an ActiveX `dispinterface` in Lua.

```
-- Creates and fills the Lua table that will implement the
-- ActiveX interface

events_table = {}

function events_table:AfterUpdate()
  print("AfterUpdate called!")
end

-- Here we implement the interface DCalendarEvents, which is part
-- of the Microsoft(R) Calendar object, whose ProgID is MSCAL.Calendar

events_obj = luacom_ImplInterface(
  events_table,
  "MSCAL.Calendar",
  "DCalendarEvents")

-- Checks for errors
--
if events_obj == nil then
  print("Implementation failed")
  exit(1)
end

-- Tests the interface: this must generate a call to the events:AfterUpdate
-- defined above
--
events_obj:AfterUpdate()
```

If the interface to be implemented is described in a stand-alone type library, the function `luacom_ImplInterfa` must be used instead:

```
-- Creates and fills the Lua table that will implement the
-- ActiveX interface

hello_table = {}

function hello:Hello()
  print("Hello World!")
end

-- Here we implement the interface IHello
--
hello_obj = luacom_ImplInterfaceFromTypelib("hello.tlb","IHello")
```

```
-- Checks for errors
--
if hello_obj == nil then
  print("Implementation failed")
  exit(1)
end

-- Tests the interface
--
hello_obj:Hello()
```

Both functions return a LuaCOM object, whose corresponding ActiveX object is implemented by the supplied table. So, any Lua calls to this LuaCOM object will be translated to ActiveX calls which, in turn, will be translated back to Lua calls on the implementation table. This LuaCOM object can be passed as an argument to ActiveX methods who expect a `dispinterface` or to LuaCOM API functions (like `luacom_addConnection`).

One can also use the `luacom_NewObject` function, which is best suited to the situation where one needs to create a complete ActiveX object in Lua and wants to export it, so that it can be accessed through COM by any running application.

2.3.2 Using Methods and Properties

The ActiveX interfaces have two "types" of members: properties and methods. LuaCOM deals with both.

Method accesses are done in the same way as calling Lua functions stored in a table and having a "self" parameter:

```
obj = luacom_CreateObject("TEST.Test")

if obj == nil then
  exit(1)
end

-- method call
a = obj:Teste(1,2)

-- another one
obj:Teste2(a+1)
```

It's important to notice the need of using the colon – ":" – for method calls. Although LuaCOM does not use the `self` parameter that Lua passes in this case, its presence is assumed, that is, LuaCOM always skips the first parameter in the case of method calls; forgetting it may cause nasty bugs.

Accessing properties is much like the same of accessing fields in Lua tables:

```
obj = luacom_CreateObject("TEST.Test")

if obj == nil then
  exit(1)
end

-- property access
a = obj.TestData

-- property setting
obj.TestData = a + 1
```

Properties may also be accessed as methods. This is mandatory when dealing with parameterized properties, that it, ones that accept (or demand) parameters. A common example of this situation is the “Item” property of collections.

```
-- property access
a = obj:TestData()

-- Parametrized property access
b = obj:TestInfo(2)

-- Accessing collections
c = obj.Files:Item(2)
```

Notice that the colon – “:” – must also be used in this situation.

When accessing properties with method calls, LuaCOM always translates the method call to a read access (property get). To set the value of a property using a method call, it’s necessary append the prefix “set”[1] to the property name and the new value must be supplied as the last argument.

```
-- property access
a = obj:TestData()

-- Setting the property
b = obj:setTestInfo(2)

-- Setting a parametrized property
c = obj.Files:setItem(2, "test.txt")
```

The prefix “get” may also be used, to clarify the code, although it’s not necessary, as the default behavior is to make a read access.

```
-- property access
a = obj:getTestData()
```

[1] In a future version it might be allowed to change the prefix.

```
b = obj:getTestInfo(2)

c = obj.Files:getItem(2)
```

Generic LuaCOM objects

To read or write properties in generic LuaCOM objects, it's necessary access them as method calls with the right prefix (get/set). The simpler semantic of table field access does not work here.

```
obj_typ = luacom_CreateObject("Some.TypedObject")
obj_untyp = luacom_CreateObject("Untyped.Object")

-- property read (get)
a = obj_typ.Value
b = obj_untyp:getValue()

-- property write (set)
obj.typ = a + 1
obj_untyp:setValue(b + 1)
```

Property Access in Lua

When implementing a COM interface in Lua, LuaCOM also supports the concept of property and of indexed properties. LuaCOM translate property reads and writes to table field accesses:

```
interface = {}

interface.Test = 1
interface.TestIndex = {2,3}

obj = luacom_ImplInterface(interface, "TEST.Test", "ITest")

-- must print "1"
print(obj.Test)

-- must print nil (if there is no member named Test2)
print(obj.Test2)

-- this writes the filed Test
obj.Test = 1

-- Indexed property read. Must return 3 (remember that
-- indexed tables start at 1 in Lua)
i = obj:TestIndex(2)

-- Sets the indexed field
obj:setTestIndex(2,4)
```

```
-- Now must return 4
i = obj:TestIndex(2)
```

2.3.3 Connection Points

The *connection points* are part of a standard ActiveX mechanism whose primary objective is to allow the ActiveX object to notify its owner of any kind of events. The connection point works as an "event sink", where events and notifications go through.

To establish a connection using LuaCOM, the owner of the ActiveX object must create a table to implement the connection interface, whose description is provided by the ActiveX object (this interface is called a *source* interface) and then call the API function `luacom_Connect`, passing as arguments the LuaCOM object for the ActiveX object and the implementation table. Doing this, LuaCOM will automatically find the default source interface, create a LuaCOM object implemented by the supplied table and then connect this object to the ActiveX object. Here follows a sample:

```
-- Creates the ActiveX object
--
calendar = luacom_CreateObject("MSCAL.Calendar")

if calendar == nil then
  exit(1)
end

-- Creates implementation table
--
calendar_events = {}

function calendar_events:AfterUpdate()
  print("Calendar updated!")
end

-- Connects object and table
--
res = luacom_Connect(calendar, calendar_events)

if res == nil then
  exit(1)
end

-- This should trigger the AfterUpdate event
--
calendar:NextMonth()
```

It's also possible to separately create a LuaCOM object implementing the connection point source interface and then connect it to the object using `luacom_AddConnection`.

```
-- Creates the ActiveX object
--
calendar = luacom_CreateObject("MSCAL.Calendar")

if calendar == nil then
  print("Error instantiating calendar")
  exit(1)
end

-- Creates implementation table
--
calendar_events = {}

function calendar_events:AfterUpdate()
  print("Calendar updated!")
end

-- Creates LuaCOM object implemented by calendar_events
--
event_handler = luacom_ImplInterface(calendar_events,
  "MSCAL.Calendar",
   "DCalendarEvents")

if event_handler == nil then
  print("Error implementing DCalendarEvents")
  exit(1)
end

-- Connects both objects
--
luacom_addConnection(calendar, event_handler)

-- This should trigger the AfterUpdate event
--
calendar:NextMonth()

-- This disconnects the connection point established
--
luacom_releaseConnection(calendar)

-- This should NOT trigger the AfterUpdate event
--
calendar:NextMonth()
```

2.3.4 Parameter Passing

LuaCOM has some policies concerning parameter passing. They specify how LuaCOM will translate COM parameter lists to Lua and vice-versa. There are two different situations to which these policies apply: calling a method of a COM object from Lua and calling a Lua function from COM. The main question here is how to deal with the different types of parameters supported by COM ("in" parameters, "out" parameters, "in-out" parameters, "optional" parameters and "defaultvalue" parameters). There is also a special policy concerning generic LuaCOM objects.

Calling COM from Lua

This situation happens when accessing a property or calling a method of a COM object through the LuaCOM object. Here follows a sample:

```
word = luacom_GetObject("Word.Application")

-- Here we are calling the "Move" method of the Application object of
-- a running instance of Microsoft(R) Word(R)
word:Move(100,100)
```

In this situation, there are two steps in the parameter passing process:

1. convert Lua parameters to COM (this will be called the "lua2com" situation);
2. convert COM's return value *and* output values back to Lua (this will be called the "com2lua" situation).

lua2com situation The translation is done based on the type information of the method (or property); it's done following the order the parameters appear in the type information of the method. The Lua parameters are used in the same order. For each parameter there are three possibilities:

The parameter is an "in" parameter LuaCOM gets the first Lua parameter not yet converted and converts it to COM using LuaCOM type conversion engine.

The parameter is an "out" parameter LuaCOM ignores this parameter, as it will only be filled by the called method. That is, the "out" parameters SHOULD NOT appear in the Lua parameter list.

The parameter is an "in-out" parameter LuaCOM does the same as for "in" parameters.

When the caller of the method wants to omit a parameter, it must pass the `nil` value; LuaCOM then proceeds accordingly, informing the called method about the omission of the parameter. If the parameter has a default value, it is used instead. Notice that LuaCOM does not complain when one omits non-optional parameters. In fact, LuaCOM ignores the fact that a parameter is or isn't optional. It leaves the responsibility for checking this to the implementation of the called method.

com2lua situation When the called method finishes, LuaCOM translates the return value and the output values (that is, the values of the "out" and "in-out" parameters) to Lua return values. That is, the method return value is returned to the Lua code as the first return value; the output values are returned in the order they appear in the parameter list (notice that here we use the Lua feature of multiple return values). If the method does not have return values, that is, is a "`void`" method, the return values will be the output values. If there are no output values either, then there will be no return values.

The called method can omit the return value or the output values; LuaCOM them will return `nil` for each omitted value.

To illustrate these concepts, here follows a sample of these situations. First, we show an excerpt of an `ODL` file describing a method of a COM object:

```
HRESULT TestShort(
  [in] short p1, // an "in" parameter
  [out] short* p2, // an "out" parameter
  [in,out] short* p3, // an "in-out" parameter
  [out,retval] short* retval); // the return value
```

Now follows a sample of what happens when calling the method:

```
-- assume that "com" is a \luacom\ object

-- Here we set p1 = 1, p3 = 2 and leave p2 uninitialized
-- When the method returns, r1 = retval and r2 = p2 and r3 = p3
r1, r2, r3 = com:TestShort(1,2)

-- WRONG! The are only two in/in-out parameters! Out parameters
-- are ignored in the lua2com parameter translation
r1, r2, r3 = com:TestShort(1,2,3) -- WRONG!

-- Here p1 = 1, p2 is uninitialized and p3 is omitted.
r1, r2, r3 = com:TestShort(1)

-- Here we ignore the output value p3
r1,r2 = com:TestShort(1)

-- Here we ignore all output values (including the return value)
com:TestShort(1,2)
```

Generic LuaCOM objects When dealing with generic LuaCOM objects, the binding adopts a different policy: all `Lua` parameters are converted to COM ones as "in-out" parameters. If the called method sets a return value, it is returned to Lua. As all parameters are set as "in-out", all of them will be returned back to Lua, modified or not by the called method.

Calling Lua from COM

This situation happens when one implements a COM `dispinterface` in Lua. The ActiveX binding has to translate the COM method calls to Lua function calls. The policy here concerning parameter list translation is the same as the one above, just exchanging "Lua" for "COM" and vice-versa. That is, all "in" an "in-out" COM parameters are translated to parameters to the Lua function call (the output parameters are ignored). When the call finishes, the first return value is translated as the return value of the COM method and the other return values are translated as the "in-out" and "out" values, following the order they appear in the method's type information. Continuing the previous example, here we show the implementation of a method callable from COM:

```
implementation = {}

-- This method receives TWO in/in-out parameters
function implementation:TestShort(p1, p2)
  -- the first one is the retval, the second the first out param
  -- the third the second out param (in fact, an in-out param)
  return p1+p2, p1-p2, p1*p2
end

-- Implements an interface
obj = luacom_ImplInterface(implementation, "TEST.Test", ITest)

-- calls the function implementation:TestShort via COM
r1, r2, r3 = obj:TestShort(1,2)
```

2.3.5 Exception Handling

COM exceptions are converted to `lua_error`'s containing the data of the exception.

2.4 Type Conversion

LuaCOM is responsible for converting values from COM to Lua and vice versa. Most of the types can be mapped from COM to Lua and vice versa without trouble. But there are some types for which the mapping is not obvious. LuaCOM then uses some predefined rules to do the type conversion. These rules must be known to avoid misinterpretation of the conversion results and to avoid errors.

2.4.1 Boolean values

Lua uses the `nil` value as false and non-`nil` values as true. As LuaCOM gives a special meaning for `nil` values in the parameter list, it can't use Lua convention for true and false values; instead, LuaCOM uses the `C` convention: the true value is a number different from zero and the false value is the number zero. Here follows a sample:

```
-- This function alters the state of the of the window.
```

```
-- state is a Lua boolean value
-- window is a LuaCOM object

function showWindow(window, state)

  if state then
    window.Visible = 1

    -- this has the same result
    windows.Visible = -10
  else
    window.Visible = 0
  end

end

-- Shows window
showWindow(window, 1)

-- Hides window
showWindow(window, nil)
```

2.4.2 Pointers to `IDispatch` and LuaCOM objects

A pointer to `IDispatch` is converted to a LuaCOMobject whose implementation is provided by this pointer. A LuaCOMobject is converted to COM simply passing its interface implementation to COM.

2.4.3 Pointers to `IUnknown`

LuaCOM just allows passing and receiving `IUnknown` pointers; it does not operate on them. They are converted from/to userdatas with a specific tag.

2.4.4 Arrays and Tables

LuaCOM converts Lua tables to SAFEARRAY's and vice-versa. To be converted, Lua tables must be "array-like", that is, all of its elements must be or "scalars" or tables of the same length. These tables must also be "array-like". Here are some samples of how is this conversion done:

Lua table	Safe Array
`table = {"name", "phone"}`	$\begin{bmatrix} "name" & "phone" \end{bmatrix}$
`table = {{1,2},{4,9}}`	$\begin{bmatrix} 1 & 2 \\ 4 & 9 \end{bmatrix}$

2.4.5 `CURRENCY` type

The `CURRENCY` values are converted to Lua as numbers. When converting a value to COM where a `CURRENCY` is expected, LuaCOM accepts both numbers and strings formatted using the current locale for currency values. Notice that this is highly dependent on the configuration and LuaCOM just uses the VARIANT conversion functions.

2.4.6 `DATE` type

When converting from COM to Lua, the `DATE` values are transformed in strings formatted according to the current locale. The converse is true: LuaCOM converts strings formatted according to the current locale to `DATE` values.

2.4.7 Error Handling

When LuaCOM cannot convert a value from or to COM it issues an exception, that may be translated to a `lua_error` or to a COM exception, depending on who is the one being called.

Chapter 3

Implementing COM objects in Lua

(This chapter is under construction. Please report bugs, mistakes or suggestions.)

3.1 Introduction

With LuaCOM it is possible to implement full-fledged COM objects using Lua. Here we understand a COM object as a composite of these parts:

- a server, which implements one or more COM objects;
- registry information, which associates a CLSID (Class ID) to a triple *server – type library – default interface*;
- a ProgID (Programmatic Identifier) which is a name associated to a CLSID;
- a type library containing a CoClass element.

The registry information maps a ProgID to a CLSID, which is, in turn, mapped to a server. The type information describes the component, that is, which interfaces it exposes and what is the default interface.

LuaCOM simplifies these tasks providing some helper functions to deal with registration and instantiation of COM servers. By now LuaCOM supports only EXE servers, although we do not see any problem in extending it to support DLL servers as well.

3.2 Is it really useful?

Some might argue that it would be better to implement COM object in languages like C++ or Visual Basic©. That's true in many situations, and false in several others. First, dealing with COM is not easy and LuaCOM hides most its complexities; besides that, there is another compelling reason for using LuaCOM at least in some situations: the semantics of Lua tables and the way LuaCOM is implemented allows one to do some neat things:

- to expose as a COM object any object that can be accessed via Lua through a table. These might be CORBA objects, C++ objects, C structures, Lua code etc. Using this feature, a legacy application or library may be "upgraded" to COM world with little extra work;

- to use COM objects anywhere a Lua table is expected. For example, a COM object might be "exported" as a CORBA object, accessible through a network;
- to add and to redefine methods of an instance of a COM object. This might be very useful in the preceding situations: an object of interest might be incremented and them exported to another client.

Of course all this flexibility comes at some cost, primarily performance. Anyway, depending on the application, the performance drawback might be negligible.

LuaCOM does not solve all problems: there is still the need of a type library, which must be build using third party tools.

3.3 Terminology

To avoid misunderstandings, here we'll supply the meaning we give to some terms used in this chapter. We don't provide formal definitions: we just want to ease the understanding of some concepts. To better understand these concepts, see COM's documentation.

Component a piece of software with some functionality that can be used by other components. It's composed by a set of objects that implement this functionality.

Component Object an object through which all the functionality of a component can be accessed, including its other objects. This object may have many interfaces.

Application Object A component object with a interface that comprises all the top-level functionality of a component; the client does not need to use other interfaces of the component object. This concept simplifies the understanding of a component, as it puts all its functionalities in an hierarchical manner (an application object together with its sub-objects, which can only be accessed through methods and properties of the application object).

COM server Some piece of code that implements one or more component objects. A COM server must tell the other applications and components which component objects it makes available. It does so *exposing* them.

CoClass A type library describing a component should have a CoClass entry, specifying some information about the component:

- a name, differentiating one CoClass from others in the same type library;
- its CLSID, the unique identifier that distinguishes this component from all others;
- the interfaces of the component object, telling which one is the default. In a typical situation, only one interface will be supplied; thus the component object could be called an Application object for that component;
- the source interface, that is, the interface the component uses to send events to the client. This interface is not implemented by the component: it just *uses* objects that implement this interface.

Lua Application Object It's the Lua table used to implement the Application Object.

3.4 Building a LuaCOM COM server

There are some steps to build a COM server using LuaCOM:

1. specify the component;
2. identify what is going to be exported: Lua application object and its sub-objects;
3. build a type library for the component;
4. define the registration information for the component;
5. register the Component object;
6. implement and expose the COM objects;
7. add COM initialization and termination code.

3.4.1 Specify the component

This is the first step: to define what functionality the component will expose. This functionality is represented by an hierarchy of objects, rooted in the Application object. Each of these objects should implement an interface.

Example Suppose we have a Lua library that implements the access of databases contained in a specific DBMS. This library has three types of objects: databases, queries and records. In COM world, this could be represented by an Application object that opens databases and returns a Database Object. A Database object has, among others, a Query method. This method receives a SQL statement and returns a Query object. The Query object is a collection, which can be iterated using the parameterized property Records, which returns an object of type Record.

3.4.2 Objects to be exported

The objects to be exported are those belonging to the hierarchy rooted in the Application object. In Lua world, objects are ordinarily represented as tables or userdatas. So it's necessary to identify (or to implement) the Lua tables used to implement the objects to be exported.

3.4.3 Building the type library

The type library should contain entries for all the interfaces of exported objects and an entry for the CoClass, specifying the interface of the Application object and the interface used to send events.

The most common way to build a type library is to write an IDL describing the type library and them use an IDL compiler, such as Microsoft's© MIDL. Notice that all the interfaces must be dispinterfaces, that is, must inherit from `IDispatch`, and must have the flag `oleautomation`.

3.4.4 Registration Information

Here we must specify the information that is used by COM to locate the component. See documentation of `luacom_RegisterObject`.

3.4.5 Registering the Component Object

Before being accessed by other applications, the component object must be registered in the system registry. This can be done with the function `luacom_RegisterObject`. This task can be simplified using the function `luacom_detectAutomation`; using this function the registration of the component can be done just running the server with the `/Register` command-line switch.

3.4.6 Implementing and Exposing the Component

Here we're dealing with COM objects implemented in Lua. Typically the COM server will call a Lua function (like `StartAutomation`) to do this task.

There are two different situations, which one demands different actions:

Implementing the Application Object Here we must use the LuaCOM function `luacom_NewObject` to create a COM object and bind it to the table of the Lua Application Object. Them this object must be made available to other applications through `luacom_ExposeObject`.

Implementing other objects The other objects of the component are obtained via the Lua Application Object as return values of functions or as values stored in the fields of the Lua Application Object (that is, via property access). These object should be implemented using `luacom_ImplInterface`. They can be implemented in the initialization (and then be stored somewhere) or can be implemented on-demand (that is, each time a COM object should be return, a call to `luacom_ImplInterface` is made).

Notice that the fields of the Lua table used to implement COM component will only be accessible if they are present in the type library. If not, they are invisible to COM.

3.4.7 Initialization and Termination

Initialization

The COM server must call the COM initialization functions (`OleInitialize` or `CoInitialize`) before LuaCOM is started. Other initialization task is the implementation and exposition of the COM objects. This task can be greatly simplified using the C/C++ LuaCOM API function `luacom_detectAutomation`.

Termination

The COM server must call (in Lua) `luacom_RevokeObject` for each exposed object. Then it must call the COM termination functions AFTER `lua_close` has been called; otherwise fatal errors may occur.

3.5 Running the COM server

A COM server built following the preceding guidelines can be used as any other COM object, that is, using `CoCreateInstance`, `CreateObject` or something like these.

3.6 Generating Events

The function `luacom_NewObject` returns a userdata that can be used to send events to clients (see chapter 5 for the reference of `luacom_NewObject`).

Chapter 4

Release Information

Here is provided miscellaneous information specific to the current version of LuaCOM. Here are recorded the current limitations of LuaCOM, its known bugs, the history of modifications since the former version, technical details etc.

4.1 Limitations

Here are listed the current limitations of LuaCOM, as of the current version, and information about future relaxation of this restrictions.

- LuaCOM currently supports only exposes COM objects as "single use" objects. That might be circumvented by exposing many times the same object. This restriction might be removed under request;
- the implementation of DLL server via LuaCOM isn't supported; this may be implemented in the next release;
- LuaCOM does not use the `IEnumVARIANT` interface for enumerations. It's necessary to use the "Item" field; this may be implemented in the next release;
- there is no "`luacom\_UnRegisterObject`" function yet. Objects registered with `luacom\_RegisterO` must be removed from the registry manually (or using another tool); this is due to the next release;
- LuaCOM does not support named parameters; it might be implemented at request;
- LuaCOM doesn't support COM methods with variable number of parameters. This could be circumvented passing the optional parameters inside a table, but this hasn't been tested. This may be implemented under request;
- there isn't support for converting tables that are not "array-like". This may be relaxed in a future version, depending on the feasibility;
- LuaCOM only allows one connection point for each ActiveX object. This limitation may be relaxed in future versions;

- it's not possible to create an instance of an ActiveX object whose initialization is done through a persistence interface (`IPersistStream`, `IPersistStorage` etc). Anyway, most of the ActiveX objects already tested initialize themselves through `CoCreateInstance`. Initialization via persistence interfaces is planned for a future release;
- LuaCOM doesn't provide access to COM interfaces that doesn't inherit from `IDispatch` interface. That is, only Automation Objects are supported. This restriction is due to the late-binding feature provided by LuaCOM. It's possible to provide access to these COM interfaces via a "proxy" Automation Object, which translate calls made through automation to vtable (early-binding) calls. It's also possible to implement this "proxy" directly using LuaCOM C/C++ API, but this hasn't been tested nor tried;
- currently, almost all exceptions generate a call to `lua_error`, possibly aborting the Lua code. Where some degree of exception handling is needed, the Lua function `call` might be used. A better exception handling mechanism might be implemented at request.

4.2 Known bugs

Here are recorded the known bugs present in LuaCOM. If any other bugs are found, please report them through LuaCOM's home page.

- LuaCOM only implements late-bound interfaces, but accepts a QueryInterface for early-bound ones. This erroneous behavior is due to the way a VB client sends events to the server. See section 4.4;
- when a table of LuaCOM objects (that is, a SAFEARRAY of `IDispatch` pointers) is passed as a parameter to a COM object, these LuaCOM objects might not be disposed automatically and may leak;
- when a COM object implemented in Lua is called from VBScript, the "in-out" parameters of type SAFEARRAY cannot be modified. If they are, VBScript will complain with a COM error.

4.3 Future Enhancements

Besides the enhancements listed in the sections 4.1 and 4.2, there are other planned enhancements:

- type-conversion "tag method", allowing the customization of the type conversion mechanism;
- better feedback when errors happen during the execution of LuaCOM API functions, besides returning `nil`;
- dynamic creation of type libraries;
- better support for creating full-fledged COM objects using Lua.

4.4 Visual Basic© issue

COM server implemented with LuaCOM can be used in VB with no trouble:

```
Public lc as Object

Set lc = CreateObject("MyCOMObject.InLuaCOM")

lc.showWindow

b = lc.getData(3)

lc.Quit
```

But if one wants to received events generated by a COM object implemented using LuaCOM, then it's necessary to use VB's `Public WithEvents`:

```
Public WithEvents obj as MyCOMObject.Application

Set obj = CreateObject("MyCOMObject.Application")

Private Sub obj_genericEvent()
  ' Put your event code here
End Sub
```

Here there is a problem: when VB assigns the result of `CreateObject` to `obj` variable, it tries to get an early bound interface (as far as I know, VB only uses late-bound interfaces with variables of type `Object`). LuaCOM does not work with early-bound interfaces (known as vtable). If you call any method using the `obj` variable, VB will throw an exception.

The solution we adopted was to accept a QueryInterface for a early-bound interface (thus allowing the use of `Public WithEvents`). Then the client *must* do a "typecast" to use correctly the COM object:

```
Public WithEvents obj_dummy as MyCOMObject.Application
Public obj as Object

Set obj_dummy = CreateObject("MyCOMObject.Application")
Set obj = obj_dummy
```

This way the client may call methods of the COM object using the `obj` variable.

4.5 History

Version 1.0

- property access modified: now parameterized properties must be accessed as functions using a prefix to differentiate property read and write. If the prefix is omitted, a property get is assumed.

- syntax "`obj.Property(param)`" is no longer supported. A colon – ":" – must be used: "`obj:Property(param)`";
- better support for implementation of COM objects, including registration and event generation;
- Type conversion engine rewritten. Now it adheres more firmly to the types specified in the type libraries;
- binding rewritten to better support "out" and "in-out" parameters and to adhere more strictly to the recommended memory allocation policies for COM;
- COM objects without type information are now supported.

Version 0.9.2

- removal of `LUACOM_TRUE` and `LUACOM_FALSE` constants; now booleans follow the same convention of the C language;
- memory and interface leaks fixed;
- some functions of the API have slightly different names;
- changes in memory allocation policy, to follow more strictly practices recommended in COM documentation;
- parameter passing policies changed;
- added limited support for `IUnknown` pointers;
- changes in type conversion;
- added limited support for implementing and registering COM objects in Lua

Version 0.9.1

- conversion to Lua 4;
- better handling of different kinds of type information (e.g. now can access Microsoft Internet Explorer© object);
- now handles more gracefully exceptions and errors;
- added support for optional parameters with default values;
- LuaCOM does not initializes COM libraries anymore; this is left to the user;
- more stringent behavior about the syntax of method calls and property access (methods with ":" and properties with ".").

Chapter 5

Reference

5.1 The C/C++ API

luacom_open

Prototype

```
  void luacom_open(lua_State* L);
```

Description

This function initializes the LuaCOM library, registering functions and tags in the given Lua state `L`. Notice that it's necessary to initialize COM before, using `OleInitialize` or `CoInitialize` or something like that.

Sample

```
int main()
{
  lua_State *L = lua_open(0);

  OleInitialize(NULL);

  luacom_open(L);

  .
  .
  .
}
```

luacom_close

Prototype

```
  void luacom_close(lua_State* L);
```

Description

This function is intended to clean up the data structures associated with LuaCOM in a specific Lua state (`L`). Currently, it does nothing, but in future releases it will do. So, do not remove from your code! It must be also called before the COM termination functions (`OleUninitialize` and `CoInitialize`) and before `lua_close`.

Sample

```
int main()
{
  lua_State *L = lua_open(0);

  OleInitialize(NULL);

  luacom_open(L);

  .
  .
  .

  luacom_close(L);

  lua_close(L);

  OleUninitialize();
}
```

luacom_detectAutomation

Prototype

```
int luacom_detectAutomation(lua_State *L, int argc, char *argv[]);
```

Description

This function gets from the top of the Lua stack a table which should hold two fields named "StartAutomation" and "Register" (these fields should contain functions that implement these actions). Then it searches the command line (provided `argc` and `argv`) for the switches "/Automation" or "/Register". If one of these switches is found, it then calls the corresponding function in the Lua table. Finally it returns a value telling what happened, so the caller function may change its course of action (if needed).

This function is simply a helper for those implementing Automation servers using LuaCOM. Most of the work should be done by the Lua code, using the functions `luacom_RegisterObject`, `luacom_NewObject`, and `luacom_ExposeObject`.

Sample

```
/*
 * com_object.cpp
 *
 * This sample C++ code initializes the libraries and
 * the COM engine to export a COM object implemented in Lua
 */

#include <ole2.h>

// libraries
extern "C"
{
#include <lua.h>
#include <lualib.h>
}

#include <luacom.h>



int main (int argc, char *argv[])
{
  int a = 0;

  CoInitialize(NULL);

  IupOpen();

  lua_State *L = lua_open(0);

  lua_baselibopen (L);
  lua_strlibopen(L);
  lua_iolibopen(L);

  luacom_open(L);

  lua_dofile(L, "implementation.lua");

  // Pushes the table containing the functions
  // responsible for the initialization of the
  // COM object

  lua_getglobal(L, "COM");
```

```
  // detects whether the program was invoked for Automation,
  // registration or none of that

  int result = luacom_detectAutomation(L, argc, argv);

  switch(result)
  {
  case LUACOM_AUTOMATION:
    // runs the message loop, as all the needed initialization
    // has already been performed
    MessageLoop();
    break;

  case LUACOM_NOAUTOMATION:
    // This only works as a COM server
    printf("Error. This is a COM server\n");
    break;

  case LUACOM_REGISTER:
    // Notifies that the COM object has been
    // registered
    printf("COM object successfully registered.");
    break;

  case LUACOM_AUTOMATION_ERROR:
    // detectAutomation found /Automation or /Register but
    // the initialization Lua functions returned some error
    printf("Error starting Automation");
    break;
  }

  luacom_close(L);
  lua_close(L);

  CoUninitialize();

  return 0;
}

-------

-- implementation.lua
--
--   This is a sample implementation of a COM server in Lua
--

-- This is the implementation of the COM object
```

```
TestObj = {}

function TestObj:showWindow()
  dialog.show()
end

function TestObj:hideWindow()
  dialog.hide()
end



-- Here we create and populate the table to
-- be used with detectAutomation

COM = {}


-- This functions creates the COM object to be
-- exported and exposes it.
function COM:StartAutomation()

  -- creates the object using its default interface

  COMAppObject, events, e = luacom_NewObject(TestObj, "TESTE.Teste")


  -- This error will be caught by detectAutomation
  if COMAppObject == nil then
     error("luacom_NewObject failed: "..e)
  end

  -- Exposes the object
  cookie = luacom_ExposeObject(COMAppObject)
  if cookie == nil then
     error("luacom_ExposeObject failed!")
  end

end



function COM:Register()

  -- fills table with registration information
  local reginfo = {}
  reginfo.VersionIndependentProgID = "TESTE.Teste"
```

```
  reginfo.ProgID = reginfo.VersionIndependentProgID..".1"
  reginfo.TypeLib = "teste.tlb"
  reginfo.CoClass = "Teste"
  reginfo.ComponentName = "Test Component"
  reginfo.Arguments = "/Automation"

  -- stores component information in the registry
  local res = luacom_RegisterObject(reginfo)
  if res == nil then
     error("luacom_RegisterObject failed!")
  end
end
```

luacom_IDispatch2LuaCOM

Prototype

```
  int luacom_IDispatch2LuaCOM(lua_State *L, void *pdisp_arg);
```

Description

This functions takes a pointer to `IDispatch`, creates a LuaCOM object for it and pushes it in the Lua stack. This function is useful when one gets an interface for a COM object from `C/C++` code and wants to use it in Lua.

Sample

```
void CreateAndExport(lua_State* L)
{
  // Creates the object
  IUnknown *obj = CreateObj();

  // Gets the IDispatch
  IDispatch* pdisp = NULL;
  QueryInterface(IID_IDISPATCH, &pdisp);

  // pushes onto lua stack
  luacom_IDispatch2LuaCOM(L, (void *) pdisp);
}
```

5.2 The Lua API

luacom_CreateObject

Use

```
luacom_obj = luacom_CreateObject(ProgID)
```

Description

This function finds the Class ID referenced by the ProgID parameter and creates an instance of the object with this Class ID. If there is any problem (ProgID not found, error instantiating object), the function returns nil.

Parameters

Parameter	Type
ProgID	String

Return Values

Return Item	Possible Values
luacom_obj	LuaCOM object nil

Sample

```
inet_obj = luacom_CreateObject("InetCtls.Inet")

if inet_obj == nil then
  print("Error! Object could not be created!")
end
```

luacom_Connect

Use

```
  implemented_obj = luacom_Connect(luacom_obj, implementation_table)
```

Description

This functions finds the default source interface of the object `luacom_obj`, creates an instance of this interface whose implementation is given by `implementation_table` and creates a connection point between the `luacom_obj` and the implemented source interface. Any calls made by the `luacom_obj` to the source interface implementation will be translated to Lua calls to member function present in the `implementation_table`. If the function succeeds, the LuaCOM object implemented by `implementation_table` is returned; otherwise, `nil` is returned.

Parameters

Parameter	Type
`luacom_obj`	LuaCOM object
`implementation_table`	Table or userdata

Return Values

Return Item	Possible Values
implemented_obj	LuaCOM object nil

Sample

```
events_handler = {}

function events_handler:NewValue(new_value)
  print(new_value)
end

events_obj = luacom_Connect(luacom_obj, events_handler)
```

luacom_ImplInterface

Use

```
implemented_obj = luacom_ImplInterface(impl_table, ProgID, interface_name)
```

Description

This function finds the type library associated with the ProgID and tries to find the type information of an interface called "interface_name". If it does, then creates an object whose implementation is "impl_table", that is, any method call or property access on this object is translated to calls or access on the members of the table. Then it makes a LuaCOM object for the implemented interface and returns it. If there are any problems in the process (ProgID not found, interface not found, interface isn't a `dispinterface`), the function returns nil.

Parameters

Parameter	Type
`impl_table`	table or userdata
`ProgID`	string
`interface_name`	string

Return Values

Return Item	Possible Values
implemented_obj	LuaCOM object nil

Sample

```
myobject = {}

function myobject:MyMethod()
  print("My method!")
end

myobject.Property = "teste"

luacom_obj = luacom_ImplInterface(myobject, "TEST.Test", "ITest")

-- these are done via Lua
myobject:MyMethod()
print(myobject.Property)

-- this call is done through COM
luacom_obj:MyMethod()
print(luacom_obj.Property)
```

luacom_ImplInterfaceFromTypelib

Use

```
impl_obj = luacom_ImplInterfaceFromTypelib(
  impl_table,
  typelib_path,
  interface_name,
  coclass_name)
```

Description

This function loads the type library whose file path is "typelib_path" and tries to find the type information of an interface called "interface_name". If it does, then creates an object whose implementation is "impl_table", that is, any method call or property access on this object is translated to calls or access on the members of the table. Then it makes a LuaCOM object for the implemented interface and returns it. If there are any problems in the process (ProgID not found, interface not found, interface isn't a `dispinterface`), the function returns nil. The "coclass_name" parameter is optional; it is only needed if the resulting LuaCOM object is to be passed to the functions `luacom_Connect`, `luacom_AddConnection` or `luacom_ExposeObject`. This parameter specifies the Component Object class name to which the interface belongs, as one interface may be used in more than one

“coclass”.

Parameters

Parameter	Type
impl_table	table or userdata
typelib_path	string
interface_name	string
coclass_name	(optional) string

Return Values

Return Item	Possible Values
implemented_obj	LuaCOM object nil

Sample

```
myobject = {}

function myobject:MyMethod()
  print("My method!")
end

myobject.Property = "teste"

luacom_obj = luacom_ImplInterfaceFromTypelib(myobject, "test.tlb",
"ITest", "Test")

-- these are done via Lua
myobject:MyMethod()
print(myobject.Property)

-- this call is done through COM
luacom_obj:MyMethod()
print(luacom_obj.Property)
```

luacom_GetObject

Use

```
luacom_obj = luacom_GetObject(ProgID)
```

Description

This function finds the Class ID referenced by the ProgID parameter and tries to find a running instance of the object having this Class ID. If there is any problem (ProgID not found, object is not running), the function returns nil.

Parameters

Parameter	Type
ProgID	String

Return Values

Return Item	Possible Values
luacom_obj	LuaCOM object nil

Sample

```
excel = luacom_GetObject("Excel.Application")

if excel  == nil then
  print("Error! Could not get object!")
end
```

luacom_NewObject

Use

```
  implemented_obj, events_sink, errmsg = luacom_NewObject(impl_table, ProgID)
```

Description

This function is analogous to `luacom_ImplInterface`, doing just a step further: it locates the default interface for the ProgID and uses its type information. That is, this function creates a Lua implementation of a COM object's default interface. This is useful when implementing a complete COM object in Lua. It also creates a connection point for sending events to the client application and returns it as the second return value. If there are any problems in the process (ProgID not found, default interface is not a `dispinterface` etc), the function returns nil twice and returns the error message as the third return value.

To send events to the client application, just call methods of the event sink table returned. The method call will be translated to COM calls to each connection. These calls may contain parameters (as specified in the type information).

Parameters

Parameter	Type
`impl_table`	table or userdata
`ProgID`	string

Return Values

Return Item	Possible Values
implemented_obj	LuaCOM object nil
event_sink	event sink table nil
errmsg	error message in the case of failure nil

Sample

```
myobject = {}

function myobject:MyMethod()
  print("My method!")
end

myobject.Property = "teste"

obj, evt, err = luacom_NewObject(myobject, "TEST.Test")

-- these are done via Lua
myobject:MyMethod()
print(myobject.Property)

-- this call is done through COM
luacom_obj:MyMethod()
print(luacom_obj.Property)

-- here we sink events
evt:Event1()
```

luacom_ExposeObject

Use

```
cookie = luacom_ExposeObject(luacom_obj)
```

Description

This function creates and registers a *class factory* for `luacom_obj`, so that other running applications can use it. It returns a cookie that must be used to unregister the object. If the function fails, it returns `nil`.

ATTENTION: the object MUST be unregistered (using `luacom_RevokeObject`) before calling `luacom_close` or `lua_close`, otherwise unhandled exceptions might occur.

Parameters

Parameter	Type
luacom_obj	LuaCOM object

Return Values

Return Item	Possible Values
cookie	number nil

Sample

```
myobject = luacom_NewObject(impl_table, "Word.Application")

cookie = luacom_ExposeObject(myobject)

function end_of_application()
  luacom_RevokeObject(cookie)
end
```

luacom_RegisterObject

Use

```
result = luacom_RegisterObject(registration_info)
```

Description

This function creates the necessary registry entries for a COM object, using the information in `registration_info` table. If the component is successfully registered, the function returns a non-nil value.

The `registration_info` table must contain the following fields[1]:

VersionIndependentProgID This field must contain a string describing the programmatic identifier for the component, e.g. “MyCompany.MyApplication”.

ProgID The same as VersionIndependentProgID but with a version number, e.g. “MyCompany.MyApplication.2”.

TypeLib The file name of the type library describing the component. This file name should contain a path, if the type library isn’t in the same folder of the executable. Samples: `mytypelib.tlb`, `c:\app\test.tlb`, `test.exe\1` (this last one can be used when the type library is bound to the executable as a resource).

CoClass The name of the component class. There must be a `coclass` entry in the type library with the same name or the registration will fail.

[1] For a better description of these fields, see COM’s documentation.

ComponentName This is the human-readable name of the component.

Arguments This field specifies what arguments will be supplied to the component executable when started via COM. Normally it should contain "`/Automation`".

This function is not a generic "registering tool" for COM components, as it assumes the component to be registered is implemented by the running executable during registration.

Parameters

Parameter	Type
registration_info	table with registration information

Return Values

Return Item	Possible Values
result	`nil` or non-`nil` value

Sample

```
-- Lua registration code

function RegisterComponent()

  reginfo.VersionIndependentProgID = "TESTE.Teste"

  -- Adds version information
  reginfo.ProgID = reginfo.VersionIndependentProgID.."".1"

  reginfo.TypeLib = "teste.tlb"
  reginfo.CoClass = "Teste"
  reginfo.ComponentName = "Test Component"
  reginfo.Arguments = "/Automation"

  local res = luacom_RegisterObject(reginfo)

  return res

end
```

luacom_addConnection

Use

```
result = luacom_addConnection(client, server)
```

Description

This function connects two LuaCOM objects, setting the `server` as an event sink for the `client`, that is, the client will call methods of the server to notify events (following the COM model). This will only work if the `client` supports connection points of the `server`'s type. If the function succeeds, it returns 1; otherwise, it returns `nil`.

Parameters

Parameter	Type
client	LuaCOM object
server	LuaCOM object

Return Values

Return Item	Possible Values
result	number nil

Sample

```
obj = luacom_CreateObject("TEST.Test")

event_sink = {}

function event_sink:KeyPress(keynumber)
  print(keynumber)
end

event_obj = luacom_ImplInterface(
              event_sink, "TEST.Test", "ITestEvents")

result = luacom_addConnection(obj, event_obj)

if result == nil then
  print("Error!")
  exit(1)
end
```

luacom_releaseConnection

Use

```
luacom_releaseConnection(client)
```

Description

This function disconnects a LuaCOM object from its event sink.

Parameters

Parameter	Type
client	LuaCOM object

Return Values

There are none.

Sample

```
obj = luacom_CreateObject("TEST.Test")

event_sink = {}

function event_sink:KeyPress(keynumber)
  print(keynumber)
end

event_obj = luacom_ImplInterface(
              event_sink, "TEST.Test", "ITestEvents")

result = luacom_addConnection(obj, event_obj)

if result == nil then
  print("Error!")
  exit(1)
end

.
.
.

luacom_releaseConnection(obj)
```

luacom_ProgIDfromCLSID

Use

```
progID = luacom_ProgIDfromCLSID(clsid)
```

Description

This function is a proxy for the Win32 function `ProgIDFromCLSID`.

Parameters

Parameter	Type
clsid	string

Return Values

Return Item	Possible Values
progID	string nil

Sample

```
progid = luacom_ProgIDfromCLSID("{8E27C92B-1264-101C-8A2F-040224009C02}")
obj = luacom_CreateObject(progid)
```

luacom_CLSIDfromProgID

Use

```
clsid = luacom_CLSIDfromProgID(progID)
```

Description

It's the inverse of `luacom_ProgIDfromCLSID`.

luacom_ShowHelp

Use

```
luacom_ShowHelp(luacom_obj)
```

Description

This function tries to locate the `luacom_obj`'s help file in its type information and shows it.

Parameters

Parameter	Type
luacom_obj	LuaCOM object

Return Values

None.

Sample

```
obj = luacom_CreateObject("TEST.Test")

luacom_ShowHelp(obj)
```

luacom_GetIUnknown

Use

```
iunknown = luacom_GetIUnknown(luacom_obj)
```

Description

This function returns a userdata holding the `IUnknown` interface pointer to the COM object behind `luacom_obj`. It's important to notice that Lua does not duplicates userdata: many calls to `luacom_GetIUnknown` for the same LuaCOM object will return the same userdata. This means that the reference count for the `IUnknown` interface will be incremented only once (that is, the first time the userdata is pushed) and will be decremented only when all the references to that userdata go out of scope (that is, when the userdata suffers garbage collection).

One possible use for this function is to check whether two LuaCOM objects reference the same COM object.

Parameters

Parameter	Type
luacom_obj	LuaCOM object

Return Values

Return Item	Possible Values
iunknown	userdata with IUnknown tag nil

Sample

```
-- Creates two LuaCOM objects for the same COM object
-- (a running instance of Microsoft Word(R)  )

word1 = luacom_GetObject("Word.Application")
word2 = luacom_GetObject("Word.Application")

-- These two userdata should be the same
unk1 = luacom_GetIUnknown(word1)
unk2 = luacom_GetIUnknown(word2)
```

```
assert(unk1 == unk2)
```

luacom_isMember

Use

```
answer = luacom_isMember(luacom_obj, member_name)
```

Description

This function returns true (that is, different from `nil`) if there exists a method or a property of the `luacom_obj` named `member_name`.

Parameters

Parameter	Type
luacom_obj	LuaCOM object
member_name	string

Return Values

Return Item	Possible Values
answer	`nil` or non-`nil`

Sample

```
obj = luacom_CreateObject("MyObject.Test")

if luacom_isMember(obj, "Test") then
  result = obj:Test()
end
```

Chapter 6

Credits

LuaCOM has been developed by Renato Cerqueira and Vinicius Almendra. The project has been sponsored by TeCGraf (Technology Group on Computer Graphics).