Next Up Previous Contents Next: Contents Contents

# **LuaCOM User Manual**

(Version 1.3b)

#### Vinicius Almendra

### Renato Cerqueira

#### **Fabio Mascarenhas**

- Contents
- 1. Introduction
  - 1.1 Features
  - 1.2 Who Should Read What (or About the Manual)
  - Some information about samples
- 2. Tutorial
  - 2.1 Using The LuaCOM library
  - 2.2 Locating COM Objects
  - 2.3 Creating Objects
  - 2.4 Getting Help about an Object
  - 2.5 Methods and Properties
  - 2.6 Releasing Objects
- 3. LuaCOM Elements
  - 3.1 LuaCOM API
  - 3.2 LuaCOM objects
    - <u>3.2.1 Object Disposal</u>
  - 3.3 Automation binding
    - 3.3.1 Implementing dispinterfaces in Lua
    - <u>3.3.2 Using Methods and Properties</u>
      - 3.3.2.1 Extensible interfaces
      - 3.3.2.2 Default methods
      - 3.3.2.3 Generic LuaCOM objects
      - 3.3.2.4 Property Access in Lua

- 3.3.3 Connection Points: handling events
- 3.3.4 Parameter Passing
  - 3.3.4.1 Calling COM from Lua
  - 3.3.4.2 Calling Lua from COM
- 3.3.5 Exception Handling
- 3.4 Type Conversion
  - 3.4.1 Boolean values
  - 3.4.2 Pointers to IDispatch and LuaCOM objects
  - 3.4.3 Pointers to IUnknown
  - 3.4.4 Arrays and Tables
  - 3.4.5 CURRENCY type
  - <u>3.4.6 DATE type</u>
  - 3.4.7 Error Handling
- 3.5 Other Objects
  - 3.5.1 The Enumerator Object
  - 3.5.2 The Connection Point Container Object
  - 3.5.3 The Typelib and Typeinfo Objects
- 4. Implementing COM objects in Lua
  - 4.1 Introduction
  - 4.2 Is it really useful?
  - 4.3 Terminology
  - <u>4.4 Building a LuaCOM COM server</u>
    - 4.4.1 Specify the component
    - 4.4.2 Objects to be exported
    - 4.4.3 Building the type library
    - 4.4.4 Registration Information
    - <u>4.4.5 Registering the Component Object</u>
    - 4.4.6 Implementing and Exposing the Component
    - 4.4.7 Initialization and Termination
      - <u>Initialization</u>
      - Termination
  - 4.5 Running the COM server
  - <u>4.6 Generating Events</u>
  - 4.7 Full Example
- 5. Release Information
  - <u>5.1 Limitations</u>
  - 5.2 Known bugs

- 5.3 Future Enhancements
- 5.4 Important issues about LuaCOM
  - <u>5.4.1 Problems instantiating COM objects</u>
  - 5.4.2 Releasing COM objects from memory
  - <u>5.4.3 Receiving events</u>
  - 5.4.4 Extensible Interfaces
  - 5.4.5 Visual Basic© issue
- 5.5 History
  - Version 1.3b
  - <u>Version 1.2</u>
  - Version 1.1
  - Version 1.0
  - Version 0.9.2
  - <u>Version 0.9.1</u>
- 6. Reference
  - 6.1 The C/C++ API
    - <u>luacom\_open</u>
      - Prototype
      - Description
      - <u>Sample</u>
    - luacom close
      - <u>Prototype</u>
      - Description
      - <u>Sample</u>
    - <u>luacom\_detectAutomation</u>
      - Prototype
      - Description
      - <u>Sample</u>
    - luacom\_IDispatch2LuaCOM
      - Prototype
      - Description
      - Sample
  - 6.2 The Lua Standard API
    - <u>CreateObject</u>
      - Use
      - <u>Description</u>
      - Parameters
      - Return Values

- Sample
- Connect
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- <u>ImplInterface</u>
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - Sample
- <u>ImplInterfaceFromTypelib</u>
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - Sample
- GetObject
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - Sample
- <u>NewObject</u>
  - <u>Use</u>
  - <u>Description</u>
  - Parameters
  - Return Values
  - Sample
- ExposeObject
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- <u>RegisterObject</u>
  - Use

- Description
- Parameters
- Return Values
- <u>Sample</u>
- RegisterObject
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- addConnection
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- <u>releaseConnection</u>
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- ProgIDfromCLSID
  - Use
  - <u>Description</u>
  - Parameters
  - Return Values
  - <u>Sample</u>
- CLSIDfromProgID
  - Use
  - Description
- ShowHelp
  - Use
  - <u>Description</u>
  - Parameters
  - Return Values
  - <u>Sample</u>
- <u>GetIUnknown</u>
  - Use

- Description
- Parameters
- Return Values
- Sample
- isMember
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- StartLog
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- EndLog
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- <u>GetEnumerator</u>
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- 6.3 Lua Extended API
  - pairs
  - <u>GetType</u>
  - CreateLocalObject
  - CreateInprocObject
  - <u>LoadConstants</u>
  - <u>FillTypeInfo</u>
  - FillTypeLib
- <u>6.4 Enumerator Object</u>
  - Methods
- <u>6.5 Type Library Object</u>

- <u>Methods</u>
- 6.6 Type Information ObjectMethods
- <u>7. Credits</u>
- About this document ...

Next Up Previous Contents Next: Some information about samples Up: 1.2 Who

**Should Read Previous: 1.2.0.0.1 Newbies Contents** 

### 1.2.0.0.2 Knowledge required

This manual presumes some knowledge of COM and Automation. We don't intend to explain in detail how these technologies work or how they can be used to solve particular problems. This information can be found easily in the web or in good books.

Next Up Previous Contents Next: 3.1 LuaCOM API Up: LuaCOM User Manual (Version Previous: 2.6 Releasing Objects Contents

# 3. LuaCOM Elements

LuaCOM is composed by the following elements:

- LuaCOM objects, which make COM objects available in Lua;
- LuaCOM API, a set of functions used to do a variety of tasks (library initialization, object creation, implementation of Automation interfaces in Lua, manipulation of connection points etc.);
- Automation binding, which translates accesses on LuaCOM objects to COM interface calls and COM 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 Automation types;
- LuaCOM parameter passing rules, which describe how LuaCOM translate a Lua parameter list to a COM one and vice versa;
- other objects, like typelib, typeinfo, enumerator etc.

#### **Subsections**

- 3.1 LuaCOM API
- 3.2 LuaCOM objects
  - 3.2.1 Object Disposal
    - **3.2.1.0.1** Caution
- 3.3 Automation binding
  - 3.3.1 Implementing dispinterfaces in Lua
  - 3.3.2 Using Methods and Properties
    - 3.3.2.1 Extensible interfaces
    - 3.3.2.2 Default methods
    - 3.3.2.3 Generic LuaCOM objects

- 3.3.2.4 Property Access in Lua
- 3.3.3 Connection Points: handling events
  - 3.3.3.0.1 Message loop
- 3.3.4 Parameter Passing
  - 3.3.4.1 Calling COM from Lua
    - 3.3.4.1.1 lua2com situation
    - 3.3.4.1.2 com2lua situation
    - 3.3.4.1.3 Generic LuaCOM objects
  - 3.3.4.2 Calling Lua from COM
- 3.3.5 Exception Handling
  - 3.3.5.0.1 Sample
- 3.4 Type Conversion
  - 3.4.1 Boolean values
    - 3.4.1.0.1 Lua 5
    - 3.4.1.0.2 Lua 4
  - 3.4.2 Pointers to IDispatch and LuaCOM objects
  - 3.4.3 Pointers to IUnknown
  - 3.4.4 Arrays and Tables
  - 3.4.5 CURRENCY type
  - <u>3.4.6 DATE type</u>
  - 3.4.7 Error Handling
- 3.5 Other Objects
  - 3.5.1 The Enumerator Object
  - 3.5.2 The Connection Point Container Object
  - 3.5.3 The Typelib and Typeinfo Objects

Next Up Previous Contents Next: 3.3 Automation binding Up: 3.2.1 Object

<u>Disposal</u> **Previous:** 3.2.1 Object Disposal **Contents** 

#### 3.2.1.0.1 Caution

LuaCOM only tracks references to COM objects. It does not work with the concepts of ``application", ``component", ``process" etc. It does not know even which objects are part of the same component or application. This has some consequences on the object disposal:

- a component may only consider as ``finished" its relationship with LuaCOM when all references to its objects are released, not only the one created with CreateObject;
- some components have a ``Quit" method. This may close the component's interface, but it could remain running if there are any references to it. Nevertheless, these references cannot be reliably used after the ``Quit" method has been called. To release the component, one must assign nil to all references to the component (and its sub-objects) and then call collectgarbage.

Next Up Previous Contents Next: 3.3.1 Implementing dispinterfaces in Up: 3.

LuaCOM Elements Previous: 3.2.1.0.1 Caution Contents

# 3.3 Automation binding

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

#### **Subsections**

- 3.3.1 Implementing dispinterfaces in Lua
- 3.3.2 Using Methods and Properties
  - 3.3.2.1 Extensible interfaces
  - 3.3.2.2 Default methods
  - 3.3.2.3 Generic LuaCOM objects
  - 3.3.2.4 Property Access in Lua
- 3.3.3 Connection Points: handling events
  - <u>3.3.3.0.1 Message loop</u>
- 3.3.4 Parameter Passing
  - 3.3.4.1 Calling COM from Lua
    - <u>3.3.4.1.1 lua2com situation</u>
    - 3.3.4.1.2 com2lua situation
    - 3.3.4.1.3 Generic LuaCOM objects
  - 3.3.4.2 Calling Lua from COM
- 3.3.5 Exception Handling
  - 3.3.5.0.1 Sample

Next Up Previous Contents Next: 3.3.2 Using Methods and Up: 3.3 Automation binding Previous: 3.3 Automation binding Contents

## 3.3.1 Implementing dispinterfaces in Lua

The Automation binding has a C++ class called tLuaDispatch 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 a dispinterface entirely in Lua, provided it has a type library describing it. This type library may be a standalone 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. LuaCOM takes care of these conversion. Follows a sample implementation of a dispinterface in Lua.

```
-- Creates and fills the Lua table that will implement the
-- COM 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.Calend
events_obj = luacom.ImplInterface(
  events_table,
  "MSCAL.Calendar"
  "DCalendarEvents")
-- Checks for errors
if events_obj == nil then
  print("Implementation failed")
  exit(1)
-- Tests the interface: this must generate a call to the events: Afte
-- defined above
events_obj:AfterUpdate()
```

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

```
-- Creates and fills the Lua table that will implement the -- Automation 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")
    os.exit(1)
end
-- Tests the interface
--
hello_obj:Hello()
```

Both methods return a LuaCOM object, whose corresponding IDispatch interface is implemented by the supplied table. This LuaCOM object can be passed as an argument to COM methods who expect a dispinterface or to LuaCOM API methods (like addConnection).

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



**Next:** 3.3.2 <u>Using Methods and</u> **Up:** 3.3 <u>Automation binding</u> **Previous:** 3.3 <u>Automation binding</u> **Contents** 

Next Up Previous Contents Next: 3.3.4 Parameter Passing Up: 3.3.3 Connection Points: handling Previous: 3.3.3 Connection Points: handling Contents

### **3.3.3.0.1** Message loop

To receive events, it is necessary to have a message loop in the thread that owns the object that is receiving the events. All events are dispatched through a Windows message queue created during COM initialization. Without a message loop, the event objects implemented by LuaCOM, will never receive method calls from the COM objects they are registered with. Out-of-process COM servers implemented with LuaCOM also need a message loop to be able to service method calls (one is provided by calling luacom.DetectAutomation).

Next Up Previous Contents Next: 3.3.4.2 Calling Lua from Up: 3.3.4.1 Calling

COM from **Previous:** 3.3.4.1.2 com2lua situation **Contents** 

### 3.3.4.1.3 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. LuaCOM assumes that these methods always return a value; if the called method does not return anything, LuaCOM pushes a nil value 3.2. As all parameters are set as ``in-out", all of them will be returned back to Lua, modified or not by the called method.

Next Up Previous Contents Next: 3.4 Type Conversion Up: 3.3.5 Exception

Handling Previous: 3.3.5 Exception Handling Contents

### 3.3.5.0.1 Sample

```
-- to make all LuaCOM errors runtime errors
luacom.config.abort_on_error = true
luacom.config.abort_on_API_error = true
-- to silently ignore all errors
luacom.config.abort_on_error = false
luacom.config.abort_on_API_error = false
-- catching an ignored error
luacom.config.last_error = nil
obj:RunMethod(x,y)
if luacom.config.last_error then
   print("Error!")
   exit(1)
end
```

All errors are also logged. Notice that some of the logged exceptions are not really errors: they are side-effects of the extensive use of exception handling inside LuaCOM code.

Next Up Previous Contents Next: 3.4.1 Boolean values Up: 3. LuaCOM Elements

Previous: 3.3.5.0.1 Sample Contents

# 3.4 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.

#### **Subsections**

- 3.4.1 Boolean values
  - o 3.4.1.0.1 Lua 5
  - o 3.4.1.0.2 Lua 4
- <u>3.4.2 Pointers to IDispatch and LuaCOM objects</u>
- 3.4.3 Pointers to IUnknown
- 3.4.4 Arrays and Tables
- 3.4.5 CURRENCY type
- <u>3.4.6 DATE type</u>
- 3.4.7 Error Handling

Next Up Previous Contents Next: 3.4.1.0.1 Lua 5 Up: 3.4 Type Conversion

Previous: 3.4 Type Conversion Contents

Contents

# 3.4.1 Boolean values

### Subsections

- 3.4.1.0.1 Lua 53.4.1.0.2 Lua 4

Next Up Previous Contents Next: 3.4.1.0.2 Lua 4 Up: 3.4.1 Boolean values

**Previous:** 3.4.1 Boolean values **Contents** 

### 3.4.1.0.1 Lua 5

LuaCOM uses the boolean values true and false, but does not works with the older convention (nil and non-nil; see paragraph below).

Next Up Previous Contents Next: 3.4.2 Pointers to IDispatch Up: 3.4.1 Boolean

<u>values</u> **Previous:** 3.4.1.0.1 Lua 5 **Contents** 

#### 3.4.1.0.2 Lua 4

This version of 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)
```

Next Up Previous Contents Next: 3.4.3 Pointers to IUnknown Up: 3.4 Type
Conversion Previous: 3.4.1.0.2 Lua 4 Contents

Contents Next: 3.4.3 Pointers to IUnknown Up: 3.4 Type

## 3.4.2 Pointers to IDispatch and LuaCOM objects

A pointer to IDispatch is converted to a LuaCOMobject whose implementation is provided by this pointer. If the object is implemented by local Lua table, then the pointer is converted to this table. A LuaCOMobject is converted to COM simply passing its interface implementation to COM.

Next Up Previous Contents Next: 3.4.5 CURRENCY type Up: 3.4 Type
Conversion Previous: 3.4.3 Pointers to IUnknown Contents

## 3.4.4 Arrays and Tables

If the table does not have a tocom tag method (for Lua 4) or \_\_tocom metamethod (for Lua 5), LuaCOM first checks if the table can be describing a date. A table is a date if it has one of those fields: Day, DayOfWeek, Month, Year, Hour, Minute, Second, Milliseconds. LuaCOM initializes the date with the fields that are present; the others are kept at their default values.

If the table is not a date, LuaCOM converts Lua tables to SAFEARRAY's and viceversa. 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"} | [ "name" "phone" ]  |
| table = {{1,2},{4,9}}     | $\left[\begin{array}{cc} 1 & 2 \\ 4 & 9 \end{array}\right]$ |

If the table has the conversion tag/metamethod, LuaCOM uses it to guide the conversion. If the tag/metamethod is a method, LuaCOM calls it, passing the table and the COM type. The method should return a COM object that LuaCOM will pass on. If the tag/metamethod is a table, LuaCOM will look for a typelib field, an interfacefield, and a coclass field, and pass those as arguments to the ImplinterfaceFromTypelib API call. If the table does not have a typelib field, LuaCOM will look for a progid field and an interface field, and pass those to the Implinterface API call. Either way, LuaCOM will pass the returned object to COM.

Next Up Previous Contents

**Next:** 3.4.5 CURRENCY type **Up:** 3.4 Type Conversion **Previous:** 3.4.3

Pointers to IUnknown Contents

# 3.4.5 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.

Next Up Previous Contents Next: 3.5.1 The Enumerator Object Up: 3. LuaCOM Elements Previous: 3.4.7 Error Handling Contents

# 3.5 Other Objects

LuaCOM deals with other objects besides COM Automation ones. Here we describe them briefly.

### **Subsections**

- 3.5.1 The Enumerator Object
- 3.5.2 The Connection Point Container Object
- 3.5.3 The Typelib and Typeinfo Objects

Next Up Previous Contents Next: 4.1 Introduction Up: LuaCOM User Manual (Version Previous: 3.5.3 The Typelib and Contents

# 4. Implementing COM objects in Lua

#### **Subsections**

- 4.1 Introduction
- 4.2 Is it really useful?
- 4.3 Terminology
- 4.4 Building a LuaCOM COM server
  - 4.4.1 Specify the component
    - 4.4.1.0.1 Example
  - 4.4.2 Objects to be exported
  - 4.4.3 Building the type library
  - 4.4.4 Registration Information
  - <u>4.4.5 Registering the Component Object</u>
  - 4.4.6 Implementing and Exposing the Component
  - 4.4.7 Initialization and Termination
    - Initialization
    - <u>Termination</u>
- 4.5 Running the COM server
- 4.6 Generating Events
- 4.7 Full Example

Next Up Previous Contents Next: 4.2 Is it really Up: 4. Implementing COM objects Previous: 4. Implementing COM objects Contents

# 4.1 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. LuaCOM suports both local (EXE) and in-process (DLL) servers.

Next Up Previous Contents Next: 4.4.1 Specify the component Up: 4.

Implementing COM objects Previous: 4.3 Terminology Contents

# 4.4 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 subobjects;
- 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.

#### **Subsections**

- 4.4.1 Specify the component
  - <u>4.4.1.0.1 Example</u>
- 4.4.2 Objects to be exported
- 4.4.3 Building the type library
- 4.4.4 Registration Information
- 4.4.5 Registering the Component Object
- 4.4.6 Implementing and Exposing the Component
- 4.4.7 Initialization and Termination
  - <u>Initialization</u>
  - Termination

Next Up Previous Contents Next: 4.4.1.0.1 Example Up: 4.4 Building a LuaCOM Previous: 4.4 Building a LuaCOM Contents

# 4.4.1 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.

#### **Subsections**

•

• 4.4.1.0.1 Example

Next Up Previous Contents Next: 4.4.2 Objects to be Up: 4.4.1 Specify the

component **Previous:** 4.4.1 Specify the component **Contents** 

#### **4.4.1.0.1** 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.

Next Up Previous Contents Next: 4.4.3 Building the type Up: 4.4 Building a LuaCOM Previous: 4.4.1.0.1 Example Contents

# 4.4.2 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.

Next Up Previous Contents Next: 5.4.2 Releasing COM objects Up: 5.4 Important issues about Previous: 5.4 Important issues about Contents

## **5.4.1 Problems instantiating COM objects**

Some COM objects can rest at in-process servers (implemented in DLL's) and at local servers (implemented as separeted processes). COM gives preference to in-process servers, as they are faster. Nevertheless, some applications may not work with LuaCOM when working as in-process servers. One should instance the COM object supplying an additional flag forcing the use of local servers. See documentation for API function CreateObject.

Next Up Previous Contents Next: 5.4.4 Extensible Interfaces Up: 5.4 Important issues about Previous: 5.4.2 Releasing COM objects Contents

# **5.4.3 Receiving events**

When one wishes to receive events and notifications from a COM object, a connection must be established using connection points. But that is not enough: the client application must have a message loop running to get these notifications. For more information, see section 3.3.3.

Next Up Previous Contents Next: Version 0.9.1 Up: 5.5 History Previous:

Version 1.0 Contents

#### 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

Next Up Previous Contents Next: Description Up: luacom\_detectAutomation

Previous: <u>luacom\_detectAutomation</u> Contents

### **Prototype**

int luacom\_detectAutomation(lua\_State \*L, int argc, char \*argv[]);

Next Up Previous Contents Next: Connect Up: CreateObject Previous: Return Values Contents

### Sample

```
inet_obj = luacom.CreateObject("InetCtls.Inet")
if inet_obj == nil then
  print("Error! Object could not be created!")
end
```



#### **Parameters**

| Parameter            | Туре              |  |
|----------------------|-------------------|--|
| luacom_obj           | LuaCOM object     |  |
| implementation_table | Table or userdata |  |

Next Up Previous Contents Next: Parameters Up: ImplInterfaceFromTypelib

**Previous:** Use Contents

#### **Description**

This method 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 method returns nil. The ``coclass\_name" parameter is optional; it is only needed if the resulting LuaCOM object is to be passed to the methods Connect, AddConnection or 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".

Next Up Previous Contents Next: GetObject Up: ImplInterfaceFromTypelib

**Previous:** Return Values Contents

#### 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)
```

Next Up Previous Contents Next: Parameters Up: ExposeObject Previous: Use Contents

#### **Description**

This method 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 method fails, it returns nil.

ATTENTION: the object MUST be unregistered (using RevokeObject) before calling luacom\_close or lua\_close, otherwise unhandled exceptions might occur.

Next Up Previous Contents Next: Description Up: RegisterObject Previous: RegisterObject Contents

#### Use

result = luacom.RegisterObject(registration\_info)

Next Up Previous Contents Next: Description Up: releaseConnection Previous: releaseConnection Contents

#### Use

luacom.releaseConnection(client, event\_sink, cookie)

Next Up Previous Contents Next: ProgIDfromCLSID Up: releaseConnection

**Previous:** Return Values Contents

#### Sample

Next Up Previous Contents Next: Description Up: ProgIDfromCLSID Previous: ProgIDfromCLSID Contents

#### Use

progID = luacom.ProgIDfromCLSID(clsid)

Next Up Previous Contents Next: Description Up: CLSIDfromProgID Previous: CLSIDfromProgID Contents

#### Use

clsid = luacom.CLSIDfromProgID(progID)

Next Up Previous Contents Next: Return Values Up: ShowHelp Previous:

Description Contents

#### **Parameters**

| Parameter  | Type          |
|------------|---------------|
| luacom_obj | LuaCOM object |

Next Up Previous Contents Next: Return Values Up: GetIUnknown Previous:

Description Contents

#### **Parameters**

| Parameter  | Туре          |
|------------|---------------|
| luacom_obj | LuaCOM object |

Next Up Previous Contents Next: isMember Up: GetIUnknown Previous: Return Values Contents

#### **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)
```

Next Up Previous Contents Next: Description Up: isMember Previous: isMember Contents

#### Use

answer = luacom.isMember(luacom\_obj, member\_name)

| Next Up Previous Contents | Next: Sample Up: EndLog Previous: | Parameters |
|---------------------------|-----------------------------------|---|
| <u>Contents</u>           |                                   |   |

### **Return Values**

None.

Next Up Previous Contents Next: 3.3.4.1 Calling COM from Up: 3.3 Automation binding Previous: 3.3.3.0.1 Message loop Contents

### 3.3.4 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.

#### **Subsections**

- 3.3.4.1 Calling COM from Lua
  - 3.3.4.1.1 lua2com situation
  - 3.3.4.1.2 com2lua situation
  - 3.3.4.1.3 Generic LuaCOM objects
- 3.3.4.2 Calling Lua from COM

Next Up Previous Contents Next: 3.3.5.0.1 Sample Up: 3.3 Automation binding Previous: 3.3.4.2 Calling Lua from Contents

## 3.3.5 Exception Handling

When a run time error occurr when using LuaCOM's methods or objects, there are two possible actions LuaCOM can take:

- to signal the error using lua\_error;
- ignore the error, just doing nothing or returning some kind of error value.

The run time errors can be divided into three types:

- errors inside API calls, like CreateObject;
- errors when using LuaCOM objects (COM method calls);
- errors inside COM objects implemented in Lua.

The third type of error is always translated into a COM exception returned to the server. To ease debugging, these errors are also logged (if the logging facility has been activated), as the server can silenty ignore these exceptions, specially in events.

If the LuaCOM library is compiled with VERBOSE defined, then a lot of informative messages are logged and all errors are displayed within a dialog box. This helps debug errors inside events on the fly, as these errors are commonly ignored by the server. Notice that this options slows down LuaCOM and can generate very big log files.

The behaviour of LuaCOM for the other two types can be customized. There is a table called config inside the LuaCOM table. This table holds three fields related to error handling:

#### abort\_on\_API\_error

if false, LuaCOM silently fails on errors inside API calls. This is NOT true for errors caused by supplying bad parameters: these always generate calls to lua\_error. The default value for this field is *false*.

#### abort on error

if false, errors inside method calls and property accesses are also ignored, possibly return nil where a return value is expected. The default value for this field is *true*.

#### last\_error

every time a run time error occurr LuaCOM sets this field with the text describing the error. This field can be used to check if some operation failed; just remember to set it to nil before the operation of interest.

#### **Subsections**

•

• 3.3.5.0.1 Sample

Next Up Previous Contents

Next: 3.3.5.0.1 Sample Up: 3.3 Automation binding Previous: 3.3.4.2 Calling

**Lua from Contents** 

## ... Lua<sup>2.1</sup>

Notice that COM must be initialized in each thread that will use it. To use LuaCOM in this situation, it's not safe to share a single Lua state among several threads; one should create a new Lua state for each thread and then initialize LuaCOM with this state.

.
.
.
.
.
.

.
.
.
.
.

·
.
.
.
.
.
.

·
·
·
·
·
·

# ... ``set"<u>3.1</u>

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

·
·
·
·
·

### ... value<sup>3.2</sup>

This feature allows a clear distinction between the return value and the inout parameters, as all parameters will end up being returned.

. . . . .

## ... parameters 5.1

Notice that LuaCOM does not implement named parameters; it just takes them when called from a COM client and puts them.

## ... fields<sup>6.1</sup>

For a better description of these fields, see COM's documentation.

## ... fields<sup>6.2</sup>

For a better description of these fields, see COM's documentation.

.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.<

. .

Next Up Previous Contents Next: 3.3.4.1.1 lua2com situation Up: 3.3.4 Parameter

Passing Previous: 3.3.4 Parameter Passing Contents

#### 3.3.4.1 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 c
-- 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).

#### **Subsections**

- <u>3.3.4.1.1 lua2com situation</u>
- 3.3.4.1.2 com2lua situation
- 3.3.4.1.3 Generic LuaCOM objects

Next Up Previous Contents Next: 3.3.4.1.2 com2lua situation Up: 3.3.4.1 Calling

COM from **Previous:** 3.3.4.1 Calling COM from **Contents** 

#### **3.3.4.1.1 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.

Next Up Previous Contents

**Next:** 3.3.4.1.2 com2lua situation **Up:** 3.3.4.1 Calling COM from **Previous:** 3.3.4.1 Calling COM from **Contents** 

Next Up Previous Contents Next: 3.3.4.1.3 Generic LuaCOM objects Up: 3.3.4.1

Calling COM from **Previous:** 3.3.4.1.1 lua2com situation **Contents** 

#### 3.3.4.1.2 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)
```

Next Up Previous Contents

Next: 3.3.4.1.3 Generic LuaCOM objects Up: 3.3.4.1 Calling COM from

**Previous:** 3.3.4.1.1 lua2com situation **Contents** 

Next Up Previous Contents Next: 3.3.5 Exception Handling Up: 3.3.4 Parameter

Passing Previous: 3.3.4.1.3 Generic LuaCOM objects Contents

#### 3.3.4.2 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)
```

Next Up Previous Next: 1. Introduction Up: LuaCOM User Manual (Version Previous: LuaCOM User Manual (Version

#### **Contents**

- 1. Introduction
  - 1.1 Features
  - 1.2 Who Should Read What (or About the Manual)
- 2. Tutorial
  - 2.1 Using The LuaCOM library
  - 2.2 Locating COM Objects
  - 2.3 Creating Objects
  - 2.4 Getting Help about an Object
  - 2.5 Methods and Properties
  - 2.6 Releasing Objects
- 3. LuaCOM Elements
  - 3.1 LuaCOM API
  - 3.2 LuaCOM objects
    - 3.2.1 Object Disposal
  - 3.3 Automation binding
    - 3.3.1 Implementing dispinterfaces in Lua
    - 3.3.2 Using Methods and Properties
    - 3.3.3 Connection Points: handling events
    - <u>3.3.4 Parameter Passing</u>
    - 3.3.5 Exception Handling
  - 3.4 Type Conversion
    - 3.4.1 Boolean values
    - 3.4.2 Pointers to IDispatch and LuaCOM objects
    - 3.4.3 Pointers to IUnknown
    - 3.4.4 Arrays and Tables
    - 3.4.5 CURRENCY type
    - <u>3.4.6 DATE type</u>
    - 3.4.7 Error Handling
  - 3.5 Other Objects
    - 3.5.1 The Enumerator Object
    - 3.5.2 The Connection Point Container Object
    - 3.5.3 The Typelib and Typeinfo Objects

#### • 4. Implementing COM objects in Lua

- 4.1 Introduction
- 4.2 Is it really useful?
- 4.3 Terminology
- 4.4 Building a LuaCOM COM server
  - 4.4.1 Specify the component
  - 4.4.2 Objects to be exported
  - 4.4.3 Building the type library
  - <u>4.4.4 Registration Information</u>
  - <u>4.4.5 Registering the Component Object</u>
  - 4.4.6 Implementing and Exposing the Component
  - 4.4.7 Initialization and Termination
- 4.5 Running the COM server
- 4.6 Generating Events
- 4.7 Full Example

#### • <u>5. Release Information</u>

- 5.1 Limitations
- 5.2 Known bugs
- 5.3 Future Enhancements
- 5.4 Important issues about LuaCOM
  - 5.4.1 Problems instantiating COM objects
  - <u>5.4.2 Releasing COM objects from memory</u>
  - <u>5.4.3 Receiving events</u>
  - 5.4.4 Extensible Interfaces
  - <u>5.4.5 Visual Basic© issue</u>
- <u>5.5 History</u>

#### • <u>6. Reference</u>

- <u>6.1 The C/C++ API</u>
- 6.2 The Lua Standard API
- 6.3 Lua Extended API
- <u>6.4 Enumerator Object</u>
- <u>6.5 Type Library Object</u>
- <u>6.6 Type Information Object</u>

#### • 7. Credits

Next Up Previous Contents Next: 1.1 Features Up: LuaCOM User Manual (Version Previous: Contents Contents

## 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 Automation technology for property access and method calls.

#### **Subsections**

- 1.1 Features
- 1.2 Who Should Read What (or About the Manual)
  - <u>1.2.0.0.1 Newbies</u>
  - 1.2.0.0.2 Knowledge required
- Some information about samples

Next Up Previous Contents Next: 1.2.0.0.1 Newbies Up: 1. Introduction
Previous: 1.1 Features Contents

# 1.2 Who Should Read What (or About the Manual)

This manual is mostly a reference manual. Here we document the behavior of LuaCOM in a variety of situations, some implementation decisions that affect the end-user of the library and its limitations. When facing some strange behavior in an application built using LuaCOM, the first step is to read all the chapter 5, where the majority of possible problems are documented. There can be found references to other sections of the manual, where more detailed information is provided.

#### **Subsections**

0

- 1.2.0.0.1 Newbies
- 1.2.0.0.2 Knowledge required

Next Up Previous Contents Next: 1.2.0.0.2 Knowledge required Up: 1.2 Who

Should Read **Previous:** 1.2 Who Should Read **Contents** 

#### **1.2.0.0.1** Newbies

For those who are newcomers, we provide a tutorial section (chapter 2) with a step-by-step guide to start using LuaCOM. More help and samples can be found in LuaCOM's home page. Notice that VBScript code can be easily converted to Lua with LuaCOM.

This manual does not provide information for developers who need deeper technical information about LuaCOM or who are willing to modify it for some reason. For this kind of information, please contact the authors.

Next Up Previous Contents Next: 2. Tutorial Up: 1. Introduction Previous: 1.2.0.0.2 Knowledge required Contents

# Some information about samples

The sample codes shown in this documentation are all for Lua 5, although most of them should also run in Lua 4. Anyway, Lua 4 specific samples can be found in the documentation for the previous version of LuaCOM.

Next Up Previous Contents Next: 2.1 Using The LuaCOM Up: LuaCOM User

Manual (Version Previous: Some information about samples Contents

# 2. Tutorial

#### **Subsections**

- 2.1 Using The LuaCOM library
- 2.2 Locating COM Objects
- 2.3 Creating Objects
- 2.4 Getting Help about an Object
- 2.5 Methods and Properties
- 2.6 Releasing Objects

Next Up Previous Contents Next: 2.2 Locating COM Objects Up: 2. Tutorial

Previous: 2. Tutorial Contents

# 2.1 Using The LuaCOM library

LuaCOM is an add-on to the Lua language. To be used, either the binary library of LuaCOM must be linked with the host program, just like the Lua library and other add-ons, or you should load a LuaCOMdynamic library through Lua 5's require/loadlib mechanism. To use dynamic loading in Lua 4 you should implement a similar mechanism. There are different versions of the LuaCOM binary for the different versions of the Lua library, so pay attention to link the right one.

If you are linking LuaCOMto your program, the next step is to modify the source code of the host program to call LuaCOM's *and* COM initialization and termination functions, which are part of the C/C++ API. To do so, include the LuaCOM's header -- luacom.h -- and call these functions in the proper order: LuaCOM must be initialize after COM and after Lua; it must be terminated before Lua; COM must be terminated AFTER Lua<sup>2.1</sup>. Here is an example of a simple c host program program using LuaCOM.

```
* Sample C program using luacom
#include <stdio.h>
#include <ole2.h> // needed for CoInitialize and CoUninitialize
#include <lua.h>
#include "luacom.h"
int main (int argc, char *argv[]) {
  /* COM initialization */
  CoInitialize(NULL);
  /* library initialization */
  lua_State *L = lua_open();
  luacom open(L);
  if(lua_dofile("luacom_sample.lua") != 0) {
    puts("Error running sample!");
    exit(1);
  luacom_close(L);
  lua_close(L);
  CoUninitialize(NULL);
  return 0;
}
```

Notice that it's necessary to initialize COM before lua\_open and to terminate it only after the last lua\_close, otherwise fatal errors may occur.

Using Lua 5 to dynamically load LuaCOM is simpler. Just call require("luacom") in your Lua script, and make sure the file luacom.lua is in your LUA\_PATH environment variable, and the Lua and LuaCOM DLLs (lua-5.0.dll, lualib-5.0.dll and luacom-lua5-1.2.dll, respectively) are in your PATH. Then run your script with the Lua standalone interpreter.



Next: 2.2 Locating COM Objects Up: 2. Tutorial Previous: 2. Tutorial

**Contents** 

Next Up Previous Contents Next: 3.2.1 Object Disposal Up: 3. LuaCOM
Elements Previous: 3.1 LuaCOM API Contents

Contents Contents

# 3.2 LuaCOM objects

LuaCOM deals with *LuaCOM objects*, which are no more than a Lua table with the LuaCOM metatable and a reference to the LuaCOM C++ object; this one is, in turn, a proxy for the COM object: it holds an IDispatch pointer to the object and translates Lua accesses to Automation 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)
```

Every time LuaCOM needs to convert an IDispatch pointer to Lua it creates a LuaCOM object. There are two situations where this happens:

- when calling LuaCOM API functions that return COM objects (CreateObject, GetObject, NewObject, Connect etc.) and
- when receiving return values from COM, where some of the values are IDispatch pointers.

Follows a sample of these situations:

```
-- First, we get a luacom object using LuaCOM API
excel = luacom.CreateObject("Excel.Application")
assert(luacomE.GetType(excel) == "LuaCOM")
-- now we get one from a method call
sheets = excel.Sheets
assert(luacomE.GetType(sheets) == "LuaCOM")
```

A LuaCOM object may be passed as a parameter 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.

#### **Subsections**

- 3.2.1 Object Disposal
  - 3.2.1.0.1 Caution

Next Up Previous Contents

Next: 3.2.1 Object Disposal Up: 3. LuaCOM Elements Previous: 3.1 LuaCOM

API **Contents** 

Next Up Previous Contents Next: 3.2.1.0.1 Caution Up: 3.2 LuaCOM objects
Previous: 3.2 LuaCOM objects
Contents

## 3.2.1 Object Disposal

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

#### **Subsections**

•

• 3.2.1.0.1 Caution

Next Up Previous Contents Next: 3.3.2.1 Extensible interfaces Up: 3.3

Automation binding Previous: 3.3.1 Implementing dispinterfaces in Contents

## 3.3.2 Using Methods and Properties

The dispinterfaces 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. Notice that this rule doesn't apply when using the default method of a LuaCOM object stored in a table or in a property of another LuaCOM object (see section 3.3.2 below).

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''\frac{3.1}{2.1} 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()
b = obj:getTestInfo(2)
c = obj.Files:getItem(2)
```

#### **Subsections**

- 3.3.2.1 Extensible interfaces
- 3.3.2.2 Default methods
- 3.3.2.3 Generic LuaCOM objects
- 3.3.2.4 Property Access in Lua

```
Next Up Previous Contents
```

**Next:** 3.3.2.1 Extensible interfaces **Up:** 3.3 Automation binding **Previous:** 3.3.1 Implementing dispinterfaces in **Contents**Fabio Mascarenhas de Queiroz 2005-01-07

Next Up Previous Contents Next: 3.3.3.0.1 Message loop Up: 3.3 Automation binding Previous: 3.3.2.4 Property Access in Contents

# 3.3.3 Connection Points: handling events

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 method 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 COM object
--
calendar = luacom.CreateObject("MSCAL.Calendar")
if calendar == nil then
    os.exit(1)
end
-- Creates implementation table
--
calendar_events = {}
function calendar_events:AfterUpdate()
    print("Calendar updated!")
end
-- Connects object and table
--
res, cookie = luacom.Connect(calendar, calendar_events)
if res == nil then
    exit(1)
end
-- This should trigger the AfterUpdate event
--
calendar:NextMonth()
```

The cookie returned by Connect identifies this connection, and can later be used to release the Connection. A COM object can have several event sinks connected to it simultaneously.

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

```
-- Instances the COM object
calendar = luacom.CreateObject("MSCAL.Calendar")
if calendar == nil then
  print("Error instantiating calendar")
  os.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)
-- Connects both objects
cookie = luacom.addConnection(calendar, event_handler)
-- This should trigger the AfterUpdate event
calendar:NextMonth()
-- This disconnects the connection point established
luacom.releaseConnection(calendar, event_handler, cookie)
-- This should NOT trigger the AfterUpdate event
calendar:NextMonth()
```

Notice that addConnection also returns a cookie. A call to releaseConnection needs both the event sink and the cookie to release the connection. The old (pre-1.3) syntax of releaseConnection (ommitting the event sink and cookie) still works, but will only release the last connection made (but there will not be leaks, all connections are released when the object is garbage-collected).

### **Subsections**

•

• 3.3.3.0.1 Message loop

Next Up Previous Contents

Next: 3.3.3.0.1 Message loop Up: 3.3 Automation binding Previous: 3.3.2.4

Property Access in **Contents** 

Next Up Previous Contents Next: Version 1.1 Up: 5.5 History Previous: Version

1.3b **Contents** 

## Version 1.2

- Can be loaded by Lua 5's require function;
- In-process servers, fully implemented in Lua (no initialization code in C is necessary for in-process servers, and for local servers using Lua 5);
- Now 1-based arrays are correctly converted by LuaCOM;
- UNICODE strings are correctly converted from/to ANSI ones by LuaCOM;
- byte arrays are now converted from/to strings with embedded zeros;
- LuaCOM now has a limited support for loading and browsing type information and type libraries. This includes the ability to import type library constants (enum's) as Lua globals and the ability to open the help information associated with a component;
- objects implementing IEnumVARIANT interface are now supported. This means that collections can be used in LuaCOM in a similar way as the are in VBScript©;
- implemented a log mechanism to simplify debugging;
- LuaCOM now handles correctly COM calls with named parameters 5.1. This caused problems when receiving Microsoft Excel© events;
- now it's possible to specify the context used to create an instance of a COM object (whether it should be created as a local server or as an in-process server);
- non-ANSI code removed;
- when faced with an IUnknown pointer, LuaCOM now queries it for IDispatch or IEnumVARIANT interfaces, returning a LuaCOM object instead of an IUnknown pointer;
- improved error-handling: now LuaCOM allows the customization of the actions to be taken when errors occur;

- LuaCOM now supports the concept of default method: when one uses a reference to a LuaCOM object as a function, LuaCOM does the function call using the default method of that object;
- part of the LuaAPI of LuaCOM now is implemented in Lua 5. This eases the addition of new features and avoids cramming the library. Nevertheless, this does not impact those who use the binary release, as they carry the Lua code precompiled;
- luacom.GetObject now supports the use of monikers. Among other thing, this makes possible to use WMI and to open document files directly, e.g. luacom.GetObject("myfile.xls");
- luacom.CreateObject and luacom.GetObject now make an attemp to initialize the COM object via IPersistStreamInit. Some objects refuse to work without this step.

Next Up Previous Contents

Next: Version 1.1 Up: 5.5 History Previous: Version 1.3b Contents

Next Up Previous Contents Next: Parameters Up: RegisterObject Previous: Use Contents

## **Description**

This method creates the necessary registry entries for a COM object, using the information in registration\_info table. If the component is successfully registered, the method returns a non-nil value.

The registration\_info table must contain the following fields 6.1:

### 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.

# 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".

# ScriptFile

This field specifies the full path of the script file that implements the component. Only used to register in-process servers.

This method 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.

Next Up Previous Contents

**Next:** Parameters **Up:** RegisterObject **Previous:** Use **Contents** *Fabio Mascarenhas de Queiroz 2005-01-07* 

Next Up Previous Contents Next: Parameters Up: RegisterObject Previous: Use Contents

### **Description**

This method removes the registry entries for a COM object, using the information in registration\_info table. If the component is successfully unregistered, the method returns a non-nil value.

The registration\_info table must contain the following fields  $\frac{6.2}{}$ :

### 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.

Next Up Previous Contents Next: Contents Contents

# **LuaCOM User Manual**

(Version 1.3b)

#### Vinicius Almendra

#### Renato Cerqueira

#### **Fabio Mascarenhas**

- Contents
- 1. Introduction
  - 1.1 Features
  - 1.2 Who Should Read What (or About the Manual)
  - Some information about samples
- 2. Tutorial
  - 2.1 Using The LuaCOM library
  - 2.2 Locating COM Objects
  - 2.3 Creating Objects
  - 2.4 Getting Help about an Object
  - 2.5 Methods and Properties
  - 2.6 Releasing Objects
- 3. LuaCOM Elements
  - 3.1 LuaCOM API
  - 3.2 LuaCOM objects
    - <u>3.2.1 Object Disposal</u>
  - 3.3 Automation binding
    - 3.3.1 Implementing dispinterfaces in Lua
    - <u>3.3.2 Using Methods and Properties</u>
      - 3.3.2.1 Extensible interfaces
      - 3.3.2.2 Default methods
      - 3.3.2.3 Generic LuaCOM objects
      - 3.3.2.4 Property Access in Lua

- 3.3.3 Connection Points: handling events
- 3.3.4 Parameter Passing
  - 3.3.4.1 Calling COM from Lua
  - 3.3.4.2 Calling Lua from COM
- 3.3.5 Exception Handling
- 3.4 Type Conversion
  - 3.4.1 Boolean values
  - 3.4.2 Pointers to IDispatch and LuaCOM objects
  - 3.4.3 Pointers to IUnknown
  - 3.4.4 Arrays and Tables
  - 3.4.5 CURRENCY type
  - <u>3.4.6 DATE type</u>
  - 3.4.7 Error Handling
- 3.5 Other Objects
  - 3.5.1 The Enumerator Object
  - 3.5.2 The Connection Point Container Object
  - 3.5.3 The Typelib and Typeinfo Objects
- 4. Implementing COM objects in Lua
  - 4.1 Introduction
  - 4.2 Is it really useful?
  - 4.3 Terminology
  - 4.4 Building a LuaCOM COM server
    - 4.4.1 Specify the component
    - 4.4.2 Objects to be exported
    - 4.4.3 Building the type library
    - 4.4.4 Registration Information
    - <u>4.4.5 Registering the Component Object</u>
    - 4.4.6 Implementing and Exposing the Component
    - 4.4.7 Initialization and Termination
      - <u>Initialization</u>
      - Termination
  - 4.5 Running the COM server
  - <u>4.6 Generating Events</u>
  - 4.7 Full Example
- 5. Release Information
  - <u>5.1 Limitations</u>
  - 5.2 Known bugs

- 5.3 Future Enhancements
- 5.4 Important issues about LuaCOM
  - <u>5.4.1 Problems instantiating COM objects</u>
  - 5.4.2 Releasing COM objects from memory
  - <u>5.4.3 Receiving events</u>
  - 5.4.4 Extensible Interfaces
  - 5.4.5 Visual Basic© issue
- 5.5 History
  - Version 1.3b
  - <u>Version 1.2</u>
  - Version 1.1
  - Version 1.0
  - Version 0.9.2
  - <u>Version 0.9.1</u>
- 6. Reference
  - 6.1 The C/C++ API
    - <u>luacom\_open</u>
      - Prototype
      - Description
      - <u>Sample</u>
    - luacom close
      - <u>Prototype</u>
      - Description
      - <u>Sample</u>
    - <u>luacom\_detectAutomation</u>
      - Prototype
      - Description
      - <u>Sample</u>
    - luacom\_IDispatch2LuaCOM
      - Prototype
      - Description
      - Sample
  - 6.2 The Lua Standard API
    - <u>CreateObject</u>
      - Use
      - <u>Description</u>
      - Parameters
      - Return Values

- Sample
- Connect
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- <u>ImplInterface</u>
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - Sample
- <u>ImplInterfaceFromTypelib</u>
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - Sample
- GetObject
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - Sample
- <u>NewObject</u>
  - <u>Use</u>
  - <u>Description</u>
  - Parameters
  - Return Values
  - Sample
- ExposeObject
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- <u>RegisterObject</u>
  - Use

- Description
- Parameters
- Return Values
- Sample
- RegisterObject
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- addConnection
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- <u>releaseConnection</u>
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- <u>ProgIDfromCLSID</u>
  - Use
  - <u>Description</u>
  - Parameters
  - Return Values
  - <u>Sample</u>
- CLSIDfromProgID
  - Use
  - Description
- ShowHelp
  - Use
  - <u>Description</u>
  - Parameters
  - Return Values
  - <u>Sample</u>
- <u>GetIUnknown</u>
  - Use

- Description
- Parameters
- Return Values
- Sample
- isMember
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- StartLog
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- EndLog
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- <u>GetEnumerator</u>
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- 6.3 Lua Extended API
  - pairs
  - <u>GetType</u>
  - <u>CreateLocalObject</u>
  - CreateInprocObject
  - <u>LoadConstants</u>
  - <u>FillTypeInfo</u>
  - FillTypeLib
- <u>6.4 Enumerator Object</u>
  - Methods
- <u>6.5 Type Library Object</u>

- <u>Methods</u>
- 6.6 Type Information ObjectMethods
- <u>7. Credits</u>
- About this document ...

Next Up Previous Contents Next: 5.1 Limitations Up: LuaCOM User Manual (Version Previous: 4.7 Full Example Contents

# 5. 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.

#### **Subsections**

- 5.1 Limitations
- 5.2 Known bugs
- 5.3 Future Enhancements
- 5.4 Important issues about LuaCOM
  - 5.4.1 Problems instantiating COM objects
  - 5.4.2 Releasing COM objects from memory
  - 5.4.3 Receiving events
  - 5.4.4 Extensible Interfaces
  - 5.4.5 Visual Basic© issue
- <u>5.5 History</u>
  - Version 1.3b
  - Version 1.2
  - Version 1.1
  - Version 1.0
  - <u>Version 0.9.2</u>
  - <u>Version 0.9.1</u>

Next Up Previous Contents Next: 6.1 The C/C++ API Up: LuaCOM User Manual (Version Previous: Version 0.9.1 Contents

# 6. Reference

#### **Subsections**

- 6.1 The C/C++ API
  - o <u>luacom\_open</u>
    - <u>Prototype</u>
    - Description
    - <u>Sample</u>
  - o <u>luacom\_close</u>
    - <u>Prototype</u>
    - Description
    - <u>Sample</u>
  - luacom detectAutomation
    - Prototype
    - Description
    - Sample
  - luacom\_IDispatch2LuaCOM
    - <u>Prototype</u>
    - Description
    - <u>Sample</u>
- 6.2 The Lua Standard API
  - CreateObject
    - <u>Use</u>
    - Description
    - Parameters
    - Return Values
    - Sample
  - Connect
    - Use
    - Description

- Parameters
- Return Values
- <u>Sample</u>
- ImplInterface
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- ImplInterfaceFromTypelib
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- GetObject
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- NewObject
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- ExposeObject
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - Sample
- RegisterObject
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample

- RegisterObject
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - Sample
- addConnection
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- releaseConnection
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- ProgIDfromCLSID
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- CLSIDfromProgID
  - <u>Use</u>
  - Description
- <u>ShowHelp</u>
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- GetIUnknown
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample

- <u>isMember</u>
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- StartLog
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- EndLog
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - Sample
- GetEnumerator
  - Use
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>
- 6.3 Lua Extended API
  - o pairs
  - GetType
  - CreateLocalObject
  - <u>CreateInprocObject</u>
  - LoadConstants
  - FillTypeInfo
  - FillTypeLib
- <u>6.4 Enumerator Object</u>
  - Methods
- <u>6.5 Type Library Object</u>
  - Methods

<u>6.6 Type Information Object</u><u>Methods</u>

Next Up Previous Contents Next: About this document ... Up: LuaCOM User Manual (Version Previous: Methods Contents

# 7. Credits

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

Next Up Previous Contents Up: LuaCOM User Manual (Version Previous: 7.
Credits Contents

# About this document ...

# **LuaCOM User Manual** (Version 1.3b)

This document was generated using the **LaTeX2HTML** translator Version 2002-2-1 (1.70)

Copyright © 1993, 1994, 1995, 1996, Nikos Drakos, Computer Based Learning Unit, University of Leeds.

Copyright © 1997, 1998, 1999, Ross Moore, Mathematics Department, Macquarie University, Sydney.

The command line arguments were:

latex2html -dir luacom-browseable -show\_section\_numbers -local\_icons
-numbered\_footnotes luacom.tex

The translation was initiated by Fabio Mascarenhas de Queiroz on 2005-01-07

Next Up Previous Contents Next: 4. Implementing COM objects Up: 3.5 Other Objects Previous: 3.5.2 The Connection Point Contents

# 3.5.3 The Typelib and Typeinfo Objects

These objects allow the navigation through the type descriptions of a LuaCOM object or of a type library. They are proxies for the interfaces ITypeLib and ITypeInfo, although not all methods are available. For more information, see sections <u>6.5</u> and <u>6.6</u>.

Next Up Previous Contents Next: Methods Up: 6. Reference Previous: Methods

**Contents** 

# **6.5 Type Library Object**

The type library object is a proxy for the interface ITypeLib. It can be obtained using the API method LoadTypeLibrary or the type information object method GetTypeLib.

#### **Subsections**

• Methods

Next Up Previous Contents Next: 6.6 Type Information Object Up: 6.5 Type Library Object Previous: 6.5 Type Library Object Contents

# Methods

#### **GetDocumentation**

returns a table containing the fields name, helpstring, helpcontext and helpfile for the type library.

## GetTypeInfoCount

returns the number of type descriptions contained in the type library.

# GetTypeInfo(n)

returns an type information object for the n-th type description.

### **ShowHelp**

tries to launch the help file associated with the type library (if any).

Next Up Previous Contents Next: Methods Up: 6. Reference Previous: Methods

**Contents** 

# **6.6 Type Information Object**

The type information object is a proxy for the interface ITypeInfo. It can be obtained using the API method GetTypeInfo or the type library object method GetTypeInfo.

#### **Subsections**

• Methods

Next Up Previous Contents Next: 7. Credits Up: 6.6 Type Information Object

Previous: 6.6 Type Information Object Contents

## **Methods**

#### **GetTypeLib**

returns the containing type library object.

### GetFuncDesc(n)

returns a table describing the n-th function of the type description. This table contains the following fields: memid (dispatch identifier), invkind (invoke kind), Params (number of parameters), ParamsOpt (number of optional parameters), description, helpfile, helpcontext, name. Besides that, it stores an array-like table called parameters describing each parameter of the function, with these fields: name, type.

#### GetVarDesc(n)

returns a table describing the n-th variable (or constant) in the type description. This table contains the following fields: name, value (for constants only).

#### GetDocumentation

returns a table with documentation for the type description, with the fields name, helpstring, helpcontext and helpfile.

## **GetTypeAttr**

returns a table containing the type attributes for the type description. This table holds the following fields: GUID, typekind, Funcs (number of functions), Vars (number of variables or constants) and ImplTypes. There is also a flags field, containing a table that describes the flags for this type description. This table contains the following boolean fields: control, appobject, dispatchable, oleautomation, cancreate.

### GetImplType(n)

For type descriptions of COM classes, this returns the type information object for the n-th interface of the COM class.

### **GetImplTypeFlags(n)**

For type descriptions of COM classes, this returns a table containing the implementation flags for the n-th interface belonging to the COM class. This table holds the following boolean flags: default, source, restricted,

defaultvtable.

### **ExportEnumerations**

returns a table with all the enumerations in this typelib. The keys are the enumeration names, and each one of them is a table, keyed by the enumeration values.



Next: 7. Credits Up: 6.6 Type Information Object Previous: 6.6 Type

**Information Object Contents** 

Next Up Previous Contents Next: Parameters Up: GetEnumerator Previous: Use Contents

## **Description**

This method returns a COM enumerator for a given LuaCOM object (if it provides one). This is the same as calling the \_NewEnum method, at least for the majority of the objects. The enumerator object is described in section <u>6.4</u>.

Next Up Previous Contents Next: pairs Up: 6. Reference Previous: Sample

**Contents** 

## 6.3 Lua Extended API

#### **Subsections**

- pairs
- <u>GetType</u>
- <u>CreateLocalObject</u>
- <u>CreateInprocObject</u>
- LoadConstants
- FillTypeInfo
- <u>FillTypeLib</u>

Next Up Previous Contents Next: FillTypeLib Up: 6.3 Lua Extended API
Previous: LoadConstants Contents

# Fill Type In fo

Next Up Previous Contents Next: 6.4 Enumerator Object Up: 6.3 Lua Extended API Previous: FillTypeInfo Contents

# ${\bf Fill Type Lib}$

Next Up Previous Contents Next: Methods Up: 6. Reference Previous:

FillTypeLib Contents

# **6.4 Enumerator Object**

The enumerator object is a proxy for the interface IEnumVARIANT. It can be obtained using the API method GetEnumerator.

#### **Subsections**

• Methods

Next Up Previous Contents Next: 6.5 Type Library Object Up: 6.4 Enumerator Object Previous: 6.4 Enumerator Object Contents

## **Methods**

#### Next

returns the next object in the enumeration or nil if the end has been reached.

## Skip

skips the next object, returning true if succeeded of false if not.

#### Reset

restarts the enumerator.

### Clone

returns a new enumerator in the same state.

Next Up Previous Contents Next: 6.3 Lua Extended API Up: GetEnumerator

**Previous:** Return Values Contents

## **Sample**

```
-- Prints all sheets of an open Excel Application
excel = luacom.GetObject("Excel.Application")
e = luacom.GetEnumerator(excel.Sheets)
s = e:Next()
while s do
   print(s.Name)
   s = e:Next()
end
```

Next Up Previous Contents Next: GetType Up: 6.3 Lua Extended API Previous: 6.3 Lua Extended API Contents

# pairs

Next Up Previous Contents Next: CreateLocalObject Up: 6.3 Lua Extended API

Previous: pairs Contents

# GetType

Next Up Previous Contents Next: CreateInprocObject Up: 6.3 Lua Extended API
Previous: GetType Contents

# ${\bf Create Local Object}$

Next Up Previous Contents Next: LoadConstants Up: 6.3 Lua Extended API
Previous: CreateLocalObject Contents

# CreateInprocObject

Next Up Previous Contents Next: FillTypeInfo Up: 6.3 Lua Extended API
Previous: CreateInprocObject Contents

## LoadConstants

Next Up Previous Contents Next: CreateObject Up: 6. Reference Previous:

Sample Contents

## 6.2 The Lua Standard API

#### **Subsections**

- <u>CreateObject</u>
  - Use
  - Description
  - Parameters
  - Return Values
  - Sample
- Connect
  - Use
  - <u>Description</u>
  - Parameters
  - Return Values
  - <u>Sample</u>
- <u>ImplInterface</u>
  - Use
  - <u>Description</u>
  - Parameters
  - Return Values
  - Sample
- ImplInterfaceFromTypelib
  - <u>Use</u>
  - <u>Description</u>
  - Parameters
  - Return Values
  - Sample
- GetObject
  - Use

- Description
- Parameters
- Return Values
- Sample

## • NewObject

- <u>Use</u>
- <u>Description</u>
- o <u>Parameters</u>
- Return Values
- Sample

## • ExposeObject

- <u>Use</u>
- Description
- Parameters
- Return Values
- Sample

### • RegisterObject

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

## • RegisterObject

- <u>Use</u>
- Description
- Parameters
- Return Values
- <u>Sample</u>

### • addConnection

- Use
- <u>Description</u>
- Parameters
- Return Values
- Sample

### • releaseConnection

- Use
- Description
- Parameters
- Return Values
- Sample

### • ProgIDfromCLSID

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- Sample

## • <u>CLSIDfromProgID</u>

- Use
- <u>Description</u>

### • ShowHelp

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

#### • GetIUnknown

- Use
- Description
- Parameters
- Return Values
- Sample

### • <u>isMember</u>

- Use
- Description
- Parameters
- Return Values
- Sample

- StartLog
  - <u>Use</u>
  - <u>Description</u>
  - Parameters
  - Return Values
  - <u>Sample</u>
- EndLog
  - <u>Use</u>
  - <u>Description</u>
  - Parameters
  - Return Values
  - Sample
- GetEnumerator
  - <u>Use</u>
  - Description
  - Parameters
  - Return Values
  - <u>Sample</u>

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

## GetEnumerator

### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Return Values Up: GetEnumerator Previous:

Description Contents

## **Parameters**

| Parameter  | Type          |
|------------|---------------|
| luacom_obj | LuaCOM object |

Next Up Previous Contents Next: Sample Up: GetEnumerator Previous:

Parameters Contents

## **Return Values**

| Return Item | Possible<br>Values          |
|-------------|-----------------------------|
| Р Р         | enumerator<br>object or nil |

Next Up Previous Contents Next: GetEnumerator Up: EndLog Previous: Return Values Contents

## Sample

luacom.EndLog()

Next Up Previous Contents Next: Description Up: GetEnumerator Previous:

GetEnumerator Contents

#### Use

e = luacom.GetEnumerator(luacom\_obj)

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# **EndLog**

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- <u>Parameters</u>
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Description Up: EndLog Previous: EndLog Contents

#### Use

luacom.EndLog()

Next Up Previous Contents Next: Parameters Up: EndLog Previous: Use Contents

## **Description**

This method stops the log facility (if it has been activated), closing the log file.

| Next Up Prev       | /ious Contents | Next: | Return | <u>Values</u> | Up: | EndLog | Previous | ;: |
|--------------------|----------------|-------|--------|---------------|-----|--------|----------|----|
| <u>Description</u> |                |       |        |               | -   |        |          |    |

### **Parameters**

None.

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

## isMember

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- <u>Parameters</u>
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Parameters Up: isMember Previous: Use Contents

### **Description**

This method returns true (that is, different from nil) if there exists a method or a property of the luacom\_obj named member\_name.

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# StartLog

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- <u>Parameters</u>
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Sample Up: StartLog Previous: Parameters Contents

### **Return Values**

| Return Item | Possible<br>Values |  |  |
|-------------|--------------------|--|--|
| result      | boolean            |  |  |

Next Up Previous Contents Next: EndLog Up: StartLog Previous: Return Values

Contents

## Sample

```
ok = luacom.StartLog("luacomlog.txt")
if not ok then
  print("log not opened")
end
```

Next Up Previous Contents Next: Parameters Up: StartLog Previous: Use Contents

#### **Description**

This methods activates the log facility of LuaCOM, writing to the log file all errors that occurr. If the library was compiled with VERBOSE defined, it also logs other informative messages like creation and destruction of LuaCOM internal objects, method calls etc. This can help track down object leaks. The method returns true if the log file could be opened, false otherwise.

Next Up Previous Contents Next: Return Values Up: StartLog Previous: Description Contents

#### **Parameters**

| Parameter     | Type   |  |
|---------------|--------|--|
| log_file_name | string |  |

Next Up Previous Contents Next: Description Up: StartLog Previous: StartLog Contents

#### Use

result = luacom.StartLog(log\_file\_name)

Next Up Previous Contents Next: Sample Up: isMember Previous: Parameters Contents

### **Return Values**

| Return Item | Possible<br>Values |  |  |
|-------------|--------------------|--|--|
| answer      | nil or non-<br>nil |  |  |

Next Up Previous Contents Next: StartLog Up: isMember Previous: Return Values Contents

### Sample

```
obj = luacom.CreateObject("MyObject.Test")
if luacom.isMember(obj, "Test") then
  result = obj:Test()
end
```

Next Up Previous Contents Next: Return Values Up: isMember Previous:

Description Contents

#### **Parameters**

| Parameter   | Туре          |  |  |  |
|-------------|---------------|--|--|--|
| luacom_obj  | LuaCOM object |  |  |  |
| member_name | string        |  |  |  |

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

## GetIUnknown

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Sample Up: GetIUnknown Previous:

Parameters Contents

### **Return Values**

| Return Item | Possible<br>Values                            |  |  |
|-------------|---|--|--|
| iunknown    | userdata with<br>IUnknown<br>metatable<br>nil |  |  |

Next Up Previous Contents Next: Description Up: GetIUnknown Previous: GetIUnknown Contents

#### Use

iunknown = luacom.GetIUnknown(luacom\_obj)

Next Up Previous Contents Next: Parameters Up: GetIUnknown Previous: Use Contents

#### **Description**

This method 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 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 method is to check whether two LuaCOM objects reference the same COM object.

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Description Contents

## ShowHelp

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- <u>Parameters</u>
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Parameters Up: ShowHelp Previous: Use Contents

## **Description**

This method tries to locate the luacom\_obj's help file in its type information and shows it.

| Next Up Previous | Contents Next: | Sample Up: | ShowHelp | <b>Previous:</b> | Parameters |
|------------------|----------------|------------|----------|------------------|------------|
| <b>Contents</b>  |                |            |          |                  |            |

### **Return Values**

None.

Next Up Previous Contents Next: GetIUnknown Up: ShowHelp Previous: Return Values Contents

## Sample

obj = luacom.CreateObject("TEST.Test")
luacom.ShowHelp(obj)

Next Up Previous Contents Next: Description Up: ShowHelp Previous: ShowHelp Contents

#### Use

luacom.ShowHelp(luacom\_obj)

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# CLSIDfromProgID

#### **Subsections**

- <u>Use</u>
- Description

Next Up Previous Contents Next: ShowHelp Up: CLSIDfromProgID Previous: Use Contents

## Description

It's the inverse of ProgIDfromCLSID.

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# ${\bf ProgID from CLSID}$

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Parameters Up: ProgIDfromCLSID Previous:
Use Contents

## **Description**

This method is a proxy for the Win32 function ProgIDFromCLSID.

Next Up Previous Contents Next: Return Values Up: ProgIDfromCLSID

Previous: Description Contents

### **Parameters**

| Parameter | Type   |
|-----------|--------|
| clsid     | string |

Next Up Previous Contents Next: Sample Up: ProgIDfromCLSID Previous:

Parameters Contents

## **Return Values**

| Return Item | Possible<br>Values |
|-------------|--------------------|
| progID      | string<br>nil      |

Next Up Previous Contents Next: CLSIDfromProgID Up: ProgIDfromCLSID

**Previous:** Return Values Contents

## Sample

progid = luacom.ProgIDfromCLSID("{8E27C92B-1264-101C-8A2F-040224009C obj = luacom.CreateObject(progid)

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# releaseConnection

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Sample Up: releaseConnection Previous:

Parameters Contents

## **Return Values**

There are none.

Next Up Previous Contents Next: releaseConnection Up: addConnection

**Previous:** Return Values Contents

### Sample

Next Up Previous Contents Next: Parameters Up: releaseConnection Previous:
Use Contents

## **Description**

This method disconnects a LuaCOM object from an event sink.

Next Up Previous Contents Next: Return Values Up: releaseConnection

**Previous:** Description Contents

## **Parameters**

| Parameter  | Туре          |
|------------|---------------|
| client     | LuaCOM object |
| event_sink | LuaCOM object |
| cookie     | LuaCOM object |

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# addConnection

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Sample Up: addConnection Previous:

Parameters Contents

## **Return Values**

| Return Item | Possible<br>Values |
|-------------|--------------------|
| cookie      | number             |

Next Up Previous Contents Next: Return Values Up: addConnection Previous:

Description Contents

## **Parameters**

| Parameter | Туре          |
|-----------|---------------|
| client    | LuaCOM object |
| server    | LuaCOM object |

Next Up Previous Contents Next: addConnection Up: RegisterObject Previous: Return Values Contents

### Sample

```
-- Lua registration code
function UnRegisterComponent()
  reginfo.VersionIndependentProgID = "TESTE.Teste"
  -- Adds version information
  reginfo.ProgID = reginfo.VersionIndependentProgID..".1"
  reginfo.TypeLib = "teste.tlb"
  reginfo.CoClass = "Teste"
  local res = luacom.UnRegisterObject(reginfo)
  return res
end
```

Next Up Previous Contents Next: Description Up: addConnection Previous: addConnection Contents

## Use

cookie = luacom.addConnection(client, server)

Next Up Previous Contents Next: Parameters Up: addConnection Previous: Use Contents

### **Description**

This method 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 method succeeds, it returns the cookie that identifies the connection; otherwise, it throws an error.

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# RegisterObject

### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# RegisterObject

### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Return Values Up: RegisterObject Previous:

Description Contents

### **Parameters**

| Parameter         | Туре                                |
|-------------------|-------------------------------------|
| registration_info | table with registration information |

Next Up Previous Contents Next: Sample Up: RegisterObject Previous: Parameters Contents

## **Return Values**

| Return Item | Possible<br>Values       |
|-------------|--------------------------|
| result      | nil or non-<br>nil value |

Next Up Previous Contents Next: Description Up: RegisterObject Previous: RegisterObject Contents

## Use

result = luacom.RegisterObject(registration\_info)



## **Return Values**

| Return Item | Possible<br>Values       |
|-------------|--------------------------|
| result      | nil or non-<br>nil value |

Next Up Previous Contents Next: RegisterObject Up: RegisterObject Previous: Return Values Contents

## 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"
  reginfo.ScriptFile = "teste.lua"
  local res = luacom.RegisterObject(reginfo)
  return res
end
```

Next Up Previous Contents Next: Return Values Up: RegisterObject Previous:

Description Contents

### **Parameters**

| Parameter         | Туре                                |
|-------------------|-------------------------------------|
| registration_info | table with registration information |

Next Up Previous Contents Next: RegisterObject Up: ExposeObject Previous: Return Values Contents

### Sample

```
myobject = luacom.NewObject(impl_table, "Word.Application")
cookie = luacom.ExposeObject(myobject)
function end_of_application()
  luacom.RevokeObject(cookie)
end
```

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# ${\bf Expose Object}$

### **Subsections**

- <u>Use</u>
- <u>Description</u>
- <u>Parameters</u>
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Description Up: ExposeObject Previous: ExposeObject Contents

## Use

cookie = luacom.ExposeObject(luacom\_obj)

Next Up Previous Contents Next: Return Values Up: ExposeObject Previous:

Description Contents

### **Parameters**

| Parameter  | Туре          |
|------------|---------------|
| luacom_obj | LuaCOM object |

Next Up Previous Contents Next: Sample Up: ExposeObject Previous: Parameters Contents

## **Return Values**

| Return Item | Possible<br>Values |
|-------------|--------------------|
| cookie      | number<br>nil      |

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# NewObject

### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Return Values Up: NewObject Previous:

Description Contents

## **Parameters**

| Parameter  | Туре              |
|------------|-------------------|
| impl_table | table or userdata |
| ProgID     | string            |

Next Up Previous Contents Next: Sample Up: NewObject Previous: Parameters Contents

## **Return Values**

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

Next Up Previous Contents Next: ExposeObject Up: NewObject Previous: Return Values Contents

#### 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()
```

Next Up Previous Contents Next: NewObject Up: GetObject Previous: Return Values Contents

# Sample

```
excel = luacom.GetObject("Excel.Application")
if excel == nil then
  print("Error! Could not get object!")
end
```

Next Up Previous Contents Next: Description Up: NewObject Previous: NewObject Contents

## Use

implemented\_obj, events\_sink, errmsg = luacom.NewObject(impl\_table

Next Up Previous Contents Next: Parameters Up: NewObject Previous: Use Contents

#### **Description**

This method is analogous to ImplInterface, doing just a step further: it locates the default interface for the ProgID and uses its type information. That is, this method 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 method 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).

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# **GetObject**

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Sample Up: GetObject Previous: Parameters Contents

# **Return Values**

| Return Item | Possible |
|-------------|----------|
|             | Values   |
| luacom_obj  | LuaCOM   |
|             | object   |
|             | nil      |

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# ImplInter face From Type lib

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Sample Up: ImplInterfaceFromTypelib

**Previous:** Parameters Contents

## **Return Values**

| Return Item     | Possible<br>Values |
|-----------------|--------------------|
| implemented_obj | LuaCOM<br>object   |
|                 | nil                |

Next Up Previous Contents Next: Description Up: GetObject Previous:

GetObject Contents

#### Use

```
luacom_obj = luacom.GetObject(ProgID)
luacom_obj = luacom.GetObject(moniker)
```

Next Up Previous Contents Next: Parameters Up: GetObject Previous: Use Contents

#### **Description**

The first version method 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 method returns nil.

The second version tries to find an object through its moniker. If there is any problem, the method returns nil.

Next Up Previous Contents Next: Return Values Up: GetObject Previous:

Description Contents

## **Parameters**

| Parameter      | Type   |
|----------------|--------|
| ProgID/moniker | String |

Next Up Previous Contents Next: Description Up: ImplInterfaceFromTypelib

**Previous:** ImplInterfaceFromTypelib Contents

#### Use

```
impl_obj = luacom.ImplInterfaceFromTypelib(
  impl_table,
  typelib_path,
  interface_name,
  coclass_name)
```

Next Up Previous Contents Next: Return Values Up: ImplInterfaceFromTypelib

Previous: Description Contents

## **Parameters**

| Parameter      | Туре              |
|----------------|-------------------|
| impl_table     | table or userdata |
| typelib_path   | string            |
| interface_name | string            |
| coclass_name   | (optional) string |

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# **ImplInterface**

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- <u>Parameters</u>
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Sample Up: ImplInterface Previous: Parameters Contents

# **Return Values**

| Return Item     | Possible<br>Values      |
|-----------------|-------------------------|
| implemented_obj | LuaCOM<br>object<br>nil |

Next Up Previous Contents Next: ImplInterfaceFromTypelib Up: ImplInterface

**Previous:** Return Values Contents

#### 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)
```

Next Up Previous Contents Next: Description Up: ImplInterface Previous: ImplInterface Contents

## Use

implemented\_obj = luacom.ImplInterface(impl\_table, ProgID, interfa

Next Up Previous Contents Next: Parameters Up: ImplInterface Previous: Use Contents

#### **Description**

This method 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 method returns nil.

Next Up Previous Contents Next: Return Values Up: ImplInterface Previous:

Description Contents

## **Parameters**

| Parameter      | Туре              |
|----------------|-------------------|
| impl_table     | table or userdata |
| ProgID         | string            |
| interface_name | string            |

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous:

Sample Contents

# **Connect**

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- <u>Parameters</u>
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Parameters Up: Connect Previous: Use Contents

#### **Description**

This method 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 method succeeds, the LuaCOM object implemented by implementation\_table, plus a cookie that identifies the connection, are returned; otherwise, nil is returned.

Notice that, to receive events, it's necessary to have a Windows message loop.

Next Up Previous Contents Next: Sample Up: Connect Previous: Parameters Contents

# **Return Values**

| Return Item     | Possible<br>Values      |
|-----------------|-------------------------|
| implemented_obj | LuaCOM<br>object<br>nil |
| cookie          | number                  |

Next Up Previous Contents Next: ImplInterface Up: Connect Previous: Return Values Contents

# **Sample**

```
events_handler = {}
function events_handler:NewValue(new_value)
  print(new_value)
end
events_obj = luacom.Connect(luacom_obj, events_handler)
```

Next Up Previous Contents Next: Use Up: 6.2 The Lua Standard Previous: 6.2

The Lua Standard Contents

# CreateObject

#### **Subsections**

- <u>Use</u>
- <u>Description</u>
- Parameters
- Return Values
- <u>Sample</u>

Next Up Previous Contents Next: Return Values Up: CreateObject Previous:

Description Contents

## **Parameters**

| Parameter | Type   |
|-----------|--------|
| ProgID    | String |

Next Up Previous Contents Next: Sample Up: CreateObject Previous: Parameters

Contents

# **Return Values**

| Return Item    | Possible |
|----------------|----------|
| Teturii Iteiii | Values   |
| luacom_obj     | LuaCOM   |
|                | object   |
|                | nil      |

Next Up Previous Contents Next: Description Up: Connect Previous: Connect Contents

## Use

implemented\_obj, cookie = luacom.Connect(luacom\_obj, implementatio

Next Up Previous Contents Next: Description Up: CreateObject Previous:

CreateObject Contents

## Use

luacom\_obj = luacom.CreateObject(ID, creation\_context, untyped)

Next Up Previous Contents Next: Parameters Up: CreateObject Previous: Use Contents

# **Description**

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

Next Up Previous Contents Next: luacom open Up: 6. Reference Previous: 6.

Reference Contents

## **6.1 The C/C++ API**

#### **Subsections**

- <u>luacom\_open</u>
  - Prototype
  - Description
  - Sample
- <u>luacom\_close</u>
  - Prototype
  - Description
  - Sample
- luacom detectAutomation
  - Prototype
  - Description
  - <u>Sample</u>
- <u>luacom\_IDispatch2LuaCOM</u>
  - <u>Prototype</u>
  - Description
  - <u>Sample</u>

Next Up Previous Contents Next: Prototype Up: 6.1 The C/C++ API Previous:

Sample Contents

# $luacom\_ID is patch 2 Lua COM$

#### **Subsections**

- <u>Prototype</u>
- <u>Description</u>
- <u>Sample</u>

Next Up Previous Contents Next: Sample Up: luacom\_IDispatch2LuaCOM

**Previous:** Prototype Contents

#### **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.

Next Up Previous Contents Next: 6.2 The Lua Standard Up: luacom IDispatch2LuaCOM Previous: Description Contents

#### 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);
}
```

Next Up Previous Contents Next: Prototype Up: 6.1 The C/C++ API Previous:

Sample Contents

## $luacom\_detect Automation$

#### **Subsections**

- <u>Prototype</u>
- <u>Description</u>
- <u>Sample</u>

Next Up Previous Contents Next: Sample Up: luacom\_detectAutomation

**Previous:** Prototype Contents

#### **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 methods RegisterObject, NewObject, and ExposeObject.

Next Up Previous Contents Next: <u>luacom\_IDispatch2LuaCOM</u> Up: luacom\_detectAutomation Previous: Description Contents

#### 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("NewObject failed: "..e)
  end
  -- Exposes the object
  cookie = luacom.ExposeObject(COMAppObject)
  if cookie == nil then
     error("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("RegisterObject failed!")
  end
end
function COM:UnRegister()
  -- fills table with registration information
  local reginfo = {}
  reginfo.VersionIndependentProgID = "TESTE.Teste"
  reginfo.ProgID = reginfo.VersionIndependentProgID..".1"
  reginfo.TypeLib = "teste.tlb"
```

reginfo.CoClass = "Teste"

```
-- removes component information from the registry
local res = luacom.UnRegisterObject(reginfo)
if res == nil then
    error("UnRegisterObject failed!")
end
end
```

Next Up Previous Contents Next: Description Up: luacom\_IDispatch2LuaCOM

Previous: luacom IDispatch2LuaCOM Contents

### **Prototype**

int luacom\_IDispatch2LuaCOM(lua\_State \*L, void \*pdisp\_arg);

Next Up Previous Contents Next: Prototype Up: 6.1 The C/C++ API Previous:

Sample Contents

## luacom\_close

#### **Subsections**

- <u>Prototype</u>
- Description
- Sample

Next Up Previous Contents Next: Description Up: luacom\_close Previous: luacom\_close Contents

## Prototype

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

Next Up Previous Contents Next: Sample Up: luacom\_close Previous: Prototype Contents

#### **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.

Next Up Previous Contents Next: <u>luacom\_detectAutomation</u> Up: <u>luacom\_close</u>

**Previous:** Description Contents

### Sample

```
int main()
{
    lua_State *L = lua_open(0);
    OleInitialize(NULL);
    luacom_open(L);
    .
    .
    .
    luacom_close(L);
    lua_close(L);
    OleUninitialize();
}
```

Next Up Previous Contents Next: Prototype Up: 6.1 The C/C++ API Previous: 6.1 The C/C++ API Contents

# luacom\_open

#### **Subsections**

- <u>Prototype</u>
- <u>Description</u>
- <u>Sample</u>

Next Up Previous Contents Next: Description Up: luacom\_open Previous: luacom\_open Contents

## Prototype

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

Next Up Previous Contents Next: Sample Up: luacom\_open Previous: Prototype

Contents

#### **Description**

This function initializes the LuaCOM library, creates the global luacom table and fills it with LuaCOM methods in the given Lua state. Notice that it's necessary to initialize COM before, using OleInitialize or CoInitialize or something like that.

```
Next Up Previous Contents Next: <u>luacom_close</u> Up: <u>luacom_open</u> Previous: <u>Description</u> Contents
```

## Sample

```
int main()
{
    lua_State *L = lua_open(0);
    OleInitialize(NULL);
    luacom_open(L);
    .
    .
    .
}
```

Next Up Previous Contents Next: Version 1.3b Up: 5. Release Information

Previous: 5.4.5 Visual Basic© issue Contents

Contents Contents

# **5.5 History**

#### **Subsections**

- <u>Version 1.3b</u>
- Version 1.2
- Version 1.1
- <u>Version 1.0</u>
- <u>Version 0.9.2</u>
- <u>Version 0.9.1</u>

Next Up Previous Contents Next: Version 0.9.2 Up: 5.5 History Previous:

Version 1.1 Contents

#### 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.

Next Up Previous Contents Next: 6. Reference Up: 5.5 History Previous: Version

<u>0.9.2</u> **Contents** 

#### 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 ``.").

Next Up Previous Contents Next: 3.2 LuaCOM objects Up: 3. LuaCOM Elements
Previous: 3. LuaCOM Elements Contents

## 3.1 LuaCOM API

The LuaCOM API is divided in two parts: the standard API and the extended API. The standard API comprises the core functionality needed to use COM objects. The extended API includes more advanced features to Lua API that simplify the development of applications using LuaCOM. This distinction has been made due to the possible unbounded growth of features, which could end up cluttering the library and making it bigger and bigger and more and more difficult to use. For now, the extended API is entirely implemented in Lua 5 and can be easily removed without trouble.

The standard API is further divided in two classes: 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. It is declared in the header file luacomh. The Lua API permits Lua programs to access all the functionality of LuaCOM. It is implemented as a set of functions inside a global table named luacom; hereafter these functions will be called LuaCOM *methods*. This table is created and populated when the C/C++ API function luacom\_open is called. Notice that the extended API lies in a different table, called luacomE. Below there is summary of the LuaCOM API. Detailed information on these methods is available in chapter 6.

| Standard Lua API |   |
|------------------|---|
| Method           | Description   |
| CreateObject     | Creates a LuaCOM object.  |
| NewObject        | Creates a LuaCOM object implemented in Lua.   |
| GetObject        | Creates a LuaCOM object associated with an instance of an already running COM object. |
| ExposeObject     | Exposes a LuaCOM object, so that other applications can get a reference to it.        |
| RevokeObject     | Undoes the operation of   |

|                          | ExposeObject.   |
|--------------------------|---|
| RegisterObject           | Fills in the registry entries necessary for exposing a COM object.  |
| UnRegisterObject         | Removes the registry entries necessary for exposing a COM object.   |
| Connect                  | Creates a connection point between an object and a Lua table.   |
| ImplInterface            | Implements an IDispatch interface using a Lua table.  |
| ImplInterfaceFromTypelib | Implements an IDispatch<br>interface described in a Type<br>Library using a Lua table.  |
| addConnection            | Connects two LuaCOM objects.  |
| releaseConnection        | Disconnects a LuaCOM object from its connection point.  |
| isMember                 | Checks whether a name correspond to a method or a property of an LuaCOM object.   |
| ProgIDfromCLSID          | Gets the ProgID associated with a CLSID.  |
| CLSIDfromProgID          | Gets the CLSID associated with a ProgID.  |
| GetIUnknown              | Returns an IUnknown interface to a LuaCOM object as a full userdata.  |
| DumpTypeInfo             | Dumps to the console the type information of the specified LuaCOM object. This method should be used only for debugging purposes. |

# Standard Lua API (continued)

| Method              | Description  |
|---------------------|--|
| GetCurrentDirectory | Returns the current directory.   |
| CreateLuaCOM        | Transforms an IUnknown full userdata into a LuaCOM object.   |
| ImportIUnknown      | Converts a light userdata<br>(pointer) to an IUnknown full<br>userdata.  |
| DetectAutomation    | Used to implement COM servers. Looks in the command-line for /Register or /UnRegister /Automation (not case-sensitive) and calls user-defined functions to register, unregister, or expose objects, entering a message loop in the latter case. If there is no command-line then assume it is being run in-process, calls the expose function and returns. |

| Extended Lua API   |  |
|--------------------|--|
| Method             | Description  |
| CreateLocalObject  | Creates a LuaCOM object as an out-of-process server.   |
| CreateInprocObject | Creates a LuaCOM object as an in-process server.   |
| ExportConstants    | Exports all the constants of a type library (standalone or bound to a LuaCOM object) to the global environment (or optionally to a table). |
| DumpTypeLib        | Creates an HTML file describing a type library.  |
| GetType            | Returns a string describing the type of object, in the case its an object belonging to the LuaCOM  |

|              | library.  |
|--------------|---|
| ViewTypeLib  | Runs DumpTypeLib and shows the created file using Internet Explorer©. |
| pairs        | Does the same as pairs for COM Enumerators.                           |
| FillTypeLib  | Creates a table describing a type library.                            |
| FillTypeInfo | Creates a table describing a type info.                               |

| Standard 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.  |

Next Up Previous Contents

Next: 3.2 LuaCOM objects Up: 3. LuaCOM Elements Previous: 3. LuaCOM

Elements **Contents** 

Next Up Previous Contents Next: 5.4.1 Problems instantiating COM Up: 5.

Release Information Previous: 5.3 Future Enhancements Contents

# 5.4 Important issues about LuaCOM

LuaCOM is very similar to using other Automation-enabled languages or environments (Visual Basic©, VBA, VBScript©, pycom etc). Nevertheless, there are some subtle differences that might confuse the programmer: different syntax, unexpected behavior etc. To ease the task of the LuaCOM user, we grouped the information related to these isses here.

#### **Subsections**

- <u>5.4.1 Problems instantiating COM objects</u>
- 5.4.2 Releasing COM objects from memory
- <u>5.4.3 Receiving events</u>
- 5.4.4 Extensible Interfaces
- 5.4.5 Visual Basic© issue

Next Up Previous Contents Next: 5.4.3 Receiving events Up: 5.4 Important issues about Previous: 5.4.1 Problems instantiating COM Contents

### **5.4.2 Releasing COM objects from memory**

In a normal scenario, an out-of-process COM server should terminate when all references to its objects are released. This may be important, as the creation of new instances might depend on the absence of a running one. LuaCOM integrates the standard COM mechanism of reference counting with Lua's garbage collection. This works fine in most situations, but there are some situations which demand a more careful analysis:

- to *immediately* terminate the server process, it's necessary to eliminate all references in Lua to the COM objects residing in this process and then force a garbage-collection cycle;
- sometimes a reference to a COM object may be stored by mistake to a
  global variable and then forgot there. This may prevent the server process to
  exit even when a method like ``Quit" is called. To avoid this problem, one
  might group all to references to a COM object and its sub-objects in a
  single table to avoid ``lost" references.

For more information, see section 3.2.1.

Next Up Previous Contents

Next: <u>5.4.3 Receiving events</u> Up: <u>5.4 Important issues about</u> Previous: <u>5.4.1</u>

Problems instantiating COM Contents

Next Up Previous Contents Next: 5.4.5 Visual Basic© issue Up: 5.4 Important issues about Previous: 5.4.3 Receiving events Contents

#### **5.4.4** Extensible Interfaces

Some objects that have type information describing their interface (methods, properties, types of parameters etc), may add new methods and properties at runtime. This means that these methods and properties can only be accessed using the same mechanism LuaCOM uses for Generic COM object. This have some implications:

- when accessing properties, it's mandatory to access them as methods and to
  use the set prefix to alter their values. If an object foo has a property color
  not present in the type information, it can only be accessed through
  foo:color() (read access) or foo:setcolor() (write access);
- when calling methods, all parameters are treated as in-out. This means that, beyond the return value, a call to a method of this type will return all the parameters back, whether or not modified by the callee. Anyway, one can ignore these values simply not assigning them to a variable, e.g. x = foo:method(a,b) will ignore the values of a and b, also returned by the call.

COM objects related to WMI have this behavior. For more information, see section <u>3.3.2</u>.

Next Up Previous Contents

Next: <u>5.4.5 Visual Basic© issue</u> Up: <u>5.4 Important issues about</u> Previous: <u>5.4.3</u>

Receiving events Contents

Next Up Previous Contents Next: Version 1.0 Up: 5.5 History Previous: Version

## 1.2 **Contents**

#### Version 1.1

- LuaCOM is now compatible with Lua 4 and Lua 5. It's just a matter of linking with the right library;
- when used with Lua 5, LuaCOM uses booleans to better match the Automation types;
- all functions of LuaCOM's Lua API are now grouped together in a single table called luacom, although they are still accessible globally as luacom\_<function> in the Lua 4 version of the library;
- now it's possible to create instances of Microsoft© Office© applications (Excel©, Powerpoint© etc.). It was only possible to use them via GetObject; now you can create a new instance of these applications using luacom.CreateObject;
- when compiled with the NDEBUG flag, LuaCOM does not use any kind of terminal output anymore (printf, cout etc). This could break some applications.

Next Up Previous Contents Next: Version 1.2 Up: 5.5 History Previous: 5.5

History **Contents** 

### **Version 1.3b**

- Conversion tag/metamethod for tables;
- Representing dates with tables;
- More than one event sink connected to an object;
- A method of typelibs returned by GetTypeInfo exports all enumerations of the typelib to a table;
- Removal of registered servers from registry (unregister);
- Identifies when an interface pointer is in fact a local Lua table implementing a COM object;
- Fixed memory leak with some out parameters;
- Removed line break in some system exceptions.

Next Up Previous Contents Next: 5.3 Future Enhancements Up: 5. Release Information Previous: 5.1 Limitations Contents

# 5.2 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 subsection <u>5.4.5</u>;
- 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 `inout" parameters of type SAFEARRAY cannot be modified. If they are, VBScript will complain with a COM error.

Next Up Previous Contents Next: 5.5 History Up: 5.4 Important issues about Previous: 5.4.4 Extensible Interfaces Contents

#### 5.4.5 Visual Basic© issue

A 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.

```
Next Up Previous Contents
```

**Next:** <u>5.5 History</u> **Up:** <u>5.4 Important issues about</u> **Previous:** <u>5.4.4 Extensible</u>

**Interfaces Contents** 

Next Up Previous Contents Next: 5. Release Information Up: 4. Implementing COM objects Previous: 4.6 Generating Events Contents

# 4.7 Full Example

This is an example of a Lua COM server. The example assumes this script is called testobj.lua:

```
-- This is the implementation of the COM object
path_to_obj = "\\Path\\To\\Script\\"
TestObj = \{\}
function TestObj:showWindow()
  print("Show!")
  events:OnShow()
end
function TestObj:hideWindow()
  print("Hide!")
  events:OnHide()
end
COM = \{\}
function COM:StartAutomation()
  -- creates the object using its default interface
  COMAppObject, events, e = luacom.NewObject(TestObj, "TEST.Test")
  -- This error will be caught by detectAutomation
  if COMAppObject == nil then
     error("NewObject failed: "..e)
  end
  -- Exposes the object
  cookie = luacom.ExposeObject(COMAppObject)
  if cookie == nil then
     error("ExposeObject failed!")
  end
end
function COM:Register()
  -- fills table with registration information
  local reginfo = {}
  reginfo.VersionIndependentProgID = "TEST.Test"
  reginfo.ProgID = reginfo.VersionIndependentProgID..".1"
  reginfo.TypeLib = "test.tlb"
  reginfo.CoClass = "Test"
  reginfo.ComponentName = "Test Component"
  reginfo.Arguments = "/Automation"
```

```
reginfo.ScriptFile = path_to_script .. "testobj.lua"
  -- stores component information in the registry
  local res = luacom.RegisterObject(reginfo)
  if res == nil then
     error("RegisterObject failed!")
  end
end
function COM:UnRegister()
  -- fills table with registration information
  local reginfo = {}
  reginfo.VersionIndependentProgID = "TEST.Test"
  reginfo.ProgID = reginfo.VersionIndependentProgID..".1"
  reginfo.TypeLib = "test.tlb"
  reginfo.CoClass = "Test"
  -- removes component information from the registry
  local res = luacom.UnRegisterObject(reginfo)
  if res == nil then
     error("UnRegisterObject failed!")
 end
end
-- Starts automation server
return luacom.DetectAutomation(COM)
```

Next Up Previous Contents Next: 5.2 Known bugs Up: 5. Release Information

Previous: 5. Release Information Contents

## 5.1 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;
- there is no ``UnRegisterObject" method yet. Objects registered with RegisterObject must be removed from the registry manually (or using another tool); this is due to the next release;
- 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;
- 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;

Next Up Previous Contents

Next: <u>5.2 Known bugs</u> Up: <u>5. Release Information</u> Previous: <u>5. Release</u>

**Information Contents** 

Next Up Previous Contents Next: 5.4 Important issues about Up: 5. Release Information Previous: 5.2 Known bugs Contents

# **5.3 Future Enhancements**

Besides the enhancements listed in the sections <u>5.1</u> and <u>5.2</u>, there are other planned enhancements:

- to improve the overall performance of LuaCOM;
- dynamic creation of type libraries;
- better support for creating full-fledged COM objects using Lua.

Next Up Previous Contents Next: Initialization Up: 4.4 Building a LuaCOM Previous: 4.4.6 Implementing and Exposing Contents

## 4.4.7 Initialization and Termination

#### **Subsections**

- <u>Initialization</u>
- <u>Termination</u>

Next Up Previous Contents Next: <u>Termination</u> **Up:** <u>4.4.7 Initialization and Termination</u> **Previous:** <u>4.4.7 Initialization and Termination</u> **Contents** 

#### **Initialization**

If you are implementing your own server, instead of using the builtin support, your 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.

If you want to use the builtin support, the only initialization necessary is to call the DetectAutomation API function at the end of the script that implements your objects, passing a table containing methods to register and expose your objects.

Next Up Previous Contents Next: 4.5 Running the COM Up: 4.4.7 Initialization and Termination Previous: Initialization Contents

#### **Termination**

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

Next Up Previous Contents Next: 4.6 Generating Events Up: 4. Implementing COM objects Previous: Termination Contents

# 4.5 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.

Next Up Previous Contents Next: 4.7 Full Example Up: 4. Implementing COM objects Previous: 4.5 Running the COM Contents

# 4.6 Generating Events

The method NewObject returns a connection point container object. This object allows the component to send events to its clients just calling methods on this object, passing the expected parameters. Return values are not allowed yet.

Next Up Previous Contents Next: 4.4.5 Registering the Component Up: 4.4

Building a LuaCOM Previous: 4.4.3 Building the type Contents

# **4.4.4 Registration Information**

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

Next Up Previous Contents Next: 4.4.6 Implementing and Exposing Up: 4.4

Building a LuaCOM Previous: 4.4.4 Registration Information Contents

## 4.4.5 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 RegisterObject API function. This function receives a table of registration info for the object. See the complete example for the fields of this table.

Next Up Previous Contents Next: 4.4.7 Initialization and Termination Up: 4.4

Building a LuaCOM Previous: 4.4.5 Registering the Component Contents

### 4.4.6 Implementing and Exposing the Component

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

#### **Implementing the Application Object**

Here we must use the LuaCOM method 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 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 Implinterface. They can be implemented in the initialization (and then be stored somewhere) or can be implemented ondemand (that is, each time a COM object should be return, a call to 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.

Next Up Previous Contents

Next: 4.4.7 Initialization and Termination Up: 4.4 Building a LuaCOM

**Previous:** 4.4.5 Registering the Component **Contents** 

Next Up Previous Contents Next: 4.4.4 Registration Information Up: 4.4 Building a LuaCOM Previous: 4.4.2 Objects to be Contents

## 4.4.3 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.

Next Up Previous Contents Next: 4.3 Terminology Up: 4. Implementing COM objects Previous: 4.1 Introduction Contents

# 4.2 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.

Next Up Previous Contents

Next: 4.3 Terminology Up: 4. Implementing COM objects Previous: 4.1

<u>Introduction</u> <u>Contents</u>

Next Up Previous Contents Next: 4.4 Building a LuaCOM Up: 4. Implementing COM objects Previous: 4.2 Is it really Contents

# 4.3 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.



Next: 4.4 Building a LuaCOM Up: 4. Implementing COM objects Previous: 4.2

<u>Is it really</u> <u>Contents</u>

Next Up Previous Contents Next: 3.5 Other Objects Up: 3.4 Type Conversion

Previous: 3.4.6 DATE type Contents

Contents Vext: 3.5 Other Objects Up: 3.4 Type Conversion

## 3.4.7 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.

Next Up Previous Contents Next: 3.5.2 The Connection Point Up: 3.5 Other Objects Previous: 3.5 Other Objects Contents

### 3.5.1 The Enumerator Object

This object is a proxy for a COM object that implements the <code>IEnumVARIANT</code> interface. It translates the calls made to fields of the table to method calls using that interface. Enumerators arise often when dealing with collections. To obtain an enumerator for a collection, use the Lua API method <code>GetEnumerator</code>. Example:

```
--
-- Sample use of enumerators
--
-- Gets an instance
word = luacom.GetObject("Word.Application")
-- Gets an enumerator for the Documents collection
docs_enum = luacom.GetEnumerator(word.Documents)
-- Prints the names of all open documents
doc = docs_enum:Next()
while doc do
   print(doc.Name)
   doc = docs_enum:Next()
end
```

The Extended Lua API method pairs allows the traversal of the enumeration using Lua's for statement. The sample above can be rewritten this way:

```
--
-- Sample use of enumerators
--
-- Gets an instance
word = luacom.GetObject("Word.Application")
-- Prints the names of all open documents
for index, doc in luacomE.pairs(word.Documents) do
   print(doc.Name)
end
```

Next Up Previous Contents Next: 3.5.3 The Typelib and Up: 3.5 Other Objects

Previous: 3.5.1 The Enumerator Object Contents

## 3.5.2 The Connection Point Container Object

This object allows a COM object implemented using LuaCOM to send events to its client. It's used primarily when implementing COM object in Lua, so see chapter 4 for more information.

Next Up Previous Contents Next: 3.4.7 Error Handling Up: 3.4 Type Conversion

Previous: 3.4.5 CURRENCY type Contents

Contents Next: 3.4.7 Error Handling Up: 3.4 Type Conversion

### **3.4.6 DATE type**

When converting from COM to Lua, the default behavior is to transform DATE values to strings formatted according to the current locale. The converse is true: LuaCOM converts strings formatted according to the current locale to DATE values.

The script can change the conversion from strings to tables by setting the DateFormat field of the luacom table (the LuaCOM namespace) to the string "table". The table will have Day, DayOfWeek, Month, Year, Hour, Minute, Second, and Milliseconds fields. To return the conversion to strings, set the DateFormat field to "string".

Next Up Previous Contents Next: 3.4.4 Arrays and Tables Up: 3.4 Type
Conversion Previous: 3.4.2 Pointers to IDispatch Contents

### 3.4.3 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 metatable.

Next Up Previous Contents Next: 3.3.2.4 Property Access in Up: 3.3.2 Using

Methods and **Previous:** 3.3.2.2 Default methods **Contents** 

### 3.3.2.3 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)
```

Next Up Previous Contents Next: 3.3.3 Connection Points: handling Up: 3.3.2

Using Methods and Previous: 3.3.2.3 Generic LuaCOM objects Contents

### 3.3.2.4 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
obi.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)
```

Next Up Previous Contents Next: 3.3.2.2 Default methods Up: 3.3.2 Using

Methods and Previous: 3.3.2 Using Methods and Contents

#### 3.3.2.1 Extensible interfaces

LuaCOM allows the use of properties as simple Lua fields just for objects that have type information. Nevertheless, some objects that *have* type information describing their interfaces implement properties that are not described in the type library: these objects implement *extensible* interfaces. Those properties can only be used with accessor functions, as shown in section 3.3.2. An example of such behaviour is found in WMI objects (Windows Management Instrumentation).

Next Up Previous Contents Next: 3.3.2.3 Generic LuaCOM objects Up: 3.3.2

Using Methods and Previous: 3.3.2.1 Extensible interfaces Contents

#### 3.3.2.2 Default methods

A dispinterface can have a default method or property, that is, one that is called when the client does not specify the method name. LuaCOM calls the default method when the object itself is used as a function.

```
excel = luacom.CreateObject("Excel.Application")
excel. Visible = true
excel.Workbooks:Add()
-- Here we call the default method
-- notice we DID NOT use the colon, as
-- the object used is Sheets, not excel
sheet = excel.Sheets(1)
print(sheet.Name)
-- Here we also call the default method
-- We must supply the self parameter
sheets = excel.Sheets
sheet2 = sheets(2)
print(sheet2.Name)
-- Setting values
excel.Sheets(1).Name = "MySheet1"
excel:Quit()
```

This can be very useful when dealing with collections, as commonly they have a default Item property.

WARNING: one must be careful not to put the colon when using default methods of LuaCOM objects contained in table or in other LuaCOM objects (see the sample above).

Next Up Previous Contents Next: 3. LuaCOM Elements Up: 2. Tutorial Previous: 2.5 Methods and Properties Contents

# 2.6 Releasing Objects

Objects are automatically released using Lua's garbage collection mechanism, that is, when there are no references to them in Lua. However, some objects may demand an explicit termination method call, like ``Quit".

```
obj = luacom.CreateObject("MyApp.MyObj")
-- Here we force an immediate release of the object
obj = nil
collectgarbage()
```

Notice that if there is any references to the COM object alive in Lua then the application (or library) that implements it will not exit.

Next Up Previous Contents Next: 2.6 Releasing Objects Up: 2. Tutorial Previous: 2.4 Getting Help about Contents

# 2.5 Methods and Properties

After creating an object, the next step is to use it. This is primarily done through *method calls* and *property accesses*. To call a method of the object, do it as if the method was a function stored in a Lua table whose key is the method name:

```
-- Here we call the method 'Show' of the COM object
myobj:Show()
-- A method with a return value
result = myobj:CheckState()
-- A method with parameters
file = myobj:LoadFile("test.xyz", 1)
-- A method with output values
x, y = myobj:UpdatePosition(x, y)
```

To read or write simple properties, one must simply use them as if they were normal table fields.

```
-- Reading properties
value1 = obj1.Value
value1 = obj2.Value
-- writing a property
obj3.Value = value1 + value2
```

Automation includes support to *parametrized properties*. These can be accessed (or written) using accessor functions.

```
value = obj:getMatrixValue(1,1)
value = value*0,125
obj:setMatrixValue(1, 1, value)
```

Next Up Previous Contents Next: 2.3 Creating Objects Up: 2. Tutorial Previous: 2.1 Using The LuaCOM Contents

# 2.2 Locating COM Objects

The first step to use a COM object is to find it. COM objects are registered in the system registry and are associated with an unique Class Identifier, known as CLSID. A CLSID may also be associated with a string known as Programmatic Identifier or ProgID. This last one is the easiest way to reference a COM object. E.g., the ProgID for Microsoft© Word© is ``word.Application''.

If one do not know in advance what is the CLSID or the ProgID of the object of interest, them it's possible to use tools like OleView to find the object, although the best place to find it is in the object's documentation.

Next Up Previous Contents Next: 2.4 Getting Help about Up: 2. Tutorial Previous: 2.2 Locating COM Objects Contents

# 2.3 Creating Objects

With the ProgID or the CLSID of an object, it's now possible to create a new instance of it or to get a running instance. To do so, the easiest way is to use the method CreateObject of the Lua API:

```
word = luacom.CreateObject("Word.Application")
assert(word)
word.Visible = true
```

If there is an already running instance of the object you want, GetObject must be used to use it. The following code illustrates this:

```
-- If there is an instance of Word(r) running,
-- it will end it
word = luacom.GetObject("Word.Application")
if word then
  word:Quit()
  word = nil
end
```

Next Up Previous Contents Next: 2.5 Methods and Properties Up: 2. Tutorial Previous: 2.3 Creating Objects Contents

# 2.4 Getting Help about an Object

To use a COM object, the first thing one must know is its *interface*, that is, its set of methods and properties. This information normally is available in the documentation of the object, but sometimes one do not have access to this documentation. LuaCOM can offer some help if the object has type information. If the object has an associated help file, LuaCOM can launch it using the method ShowHelp:

```
word = luacom.CreateObject("Word.Application")
assert(word)
luacom.ShowHelp(word)
```

If the object has an associated type library, LuaCOM can generate and display an HTML file describing it. This information can also be read using other type library browsers, as OleView.

The method DumpTypeInfo can be used in console applications to list the methods and properties of the interface. It does not give much information, but can be helpful when playing with an object.

Next Up Previous Contents Next: 1.2 Who Should Read Up: 1. Introduction

Previous: 1. Introduction Contents

Contents

## 1.1 Features

Currently, the LuaCOM library supports the following features:

- dynamic instantiation of COM objects registered in the system registry, via the CreateObject method;
- dynamic access to running COM objects via GetObject;
- COM method calls as normal Lua function calls and property accesses as normal table field accesses;
- ability to read type libraries and to generate HTML documentation on-thefly for COM objects;
- use of COM objects without type information;
- 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 5 and with Lua 4;
- log mechanism to ease the debugging of applications.