Comet

version 1.1

Typeset in LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

Contents

Chapter 1

Comet User's Guide

Comet is an Erlang-to-COM port driver. It enables Erlang to call COM objects under Windows.

1.1 Introduction

1.1.1 Introduction

In Windows, many applications and libraries call each other using COM. COM, short for Component Object Model, is Microsoft's technique for component software and distributed computing. COM enables code to be called across language boundaries. For instance, a Visual Basic program can call modules and classes written in C++, and vice versa.

Comet is a module for Erlang, that enables Erlang programs to call COM objects.

This document describes Comet and its possible applications. However, some knowledge of COM is assumed. Please check the list of recommended reading below for tips on books on COM.

1.1.2 Architecture

Comet is a module that consists of two parts: a genserver, called erl_com, that implements the Erlang part of Comet, and a port driver erl_com_drv, that implements the C glue. The port driver is also available as a port program with the same API from Erlang.

Since COM is a Windows-specific technology, Comet is only available on Windows.

Erlang can only call other COM-servers, Comet does not currently provide for creating COM-objects written in Erlang. This might be included in later versions. Thus Comet is a COM-client only.

1.1.3 Prerequisites

Basic knowledge of COM and Windows programming is assumed.

1.1.4 Where to Find More Information

More information on COM can be found in:

1. Microsoft documentation http://www.microsoft.com/com/[1]
2. Inside COM, by Dale Rogerson (ISBN: 1572313498)
3. Essential COM, by Don Box (ISBN: 0201634465)
4. Mastering COM and COM+, by Rofail and Shohoud (ISBN: 0782123848)

1.2 Comet

1.2.1 Basic COM from Erlang

In COM, there are interfaces. An interface is a handle to an object. Physically it consists of a pointer to an object with a method table.

Interfaces in COM are represented as a tuple in Erlang. This tuple should be considered oblique data.

There is a standard set of types in COM. These types can be converted to and from Erlang by the port program. (It is actually converted from the Erlang binary format.) Table 1 shows the types in Erlang and their corresponding COM type.

`integer`	`VT_I4`, 32-bits integer
`string`	`VT_STR`, unicode string
`atom`	no type, however the two atoms`true` and `false` are converted to `VT_BOOL`, the COM Boolean type
`float`	`VT_R8`, 64-bits floating point

Table 1.1: Erlang Types and Their Corresponding COM Type

However, there are fewer types in Erlang than in COM, so some types overlap. When calling a COM function, the parameters types must match exactly, since COM is strongly typed. Comet uses a special notation for the conversion of Erlang types to COM types, a tuple with an atom followed by the value, e.g. `{vt_i2, 12}`.

The comet application consists of two parts: the `gen_server` module `erl_com`, that implements the Erlang API for comet, and the port (driver or program) `erl_com_drv`, that communicates with COM objects. The port is never called directly, only through API functions in `erl_com`.

There is also a module `com_gen` that can generate Erlang APIs for COM object, by querying their type libraries.

1.2.2 How comet works

TBD

[1] URL: `http://www.microsoft.com/com/`

1.2.3 Using objects and interfaces

TBD

1.2.4 Using the dispatch interface

The dispatch or `IDispatch` interface is a way for scripts to call functions. It is used by Visual Basic, JScript and other scripting language. It is sometimes referred to as the late-binding call interface.

This way to call COM objects shows off its VB heritage. An interface has methods and properties. A property is really two methods: put property and get property.

In the `erl_com` server, there are three functions for calling an IDispatch-interface.

`invoke(Interface, Method, Parameterlist)`	Invokes a normal COM method. A list of out-parameters are returned, or, if there is a retval-parameter, it is returned.
`property_put(Interface, Method, Parameterlist, Value)`	Calls a COM method with the propput-attribute. An extra argument, after the Parameterlist, contains the property value to set. (Which really is just a parameter to the function.) If the property does not have parameters, the parameterlist might be omitted, but a value must always be provided.
`property_put_ref(Interface, Method, Parameterlist, Value)`	Sets a property value REF. See COM documentation for more info.
`property_get(Interface, Method, Parameterlist)`	Calls a COM method with the propget-attribute. The value of the property is returned. If the property does not have parameters, the parameterlist might be omitted.

Table 1.2: Functions for dispatch interfaces

The parameter `Method` above, is either a string or a member ID.

Examples of using this:

```
erl_com:invoke(Obj, "Navigate", ["www.erlang.org"])
```

```
erl_com:property_put(Obj, "Visible", true)
```

Here we assume that `Obj` is an IWebBrowser interface for an Internet Explorer program.

Calling methods this way is OK for testing things in an Erlang shell, but it's not very practical and does not make for readable code. A much simpler way of using COM objects is to generate them first and then call them.

1.2.5 Other functions

TBD

1.2.6 Generating stub modules

The `com_gen` erlang module, has functions for generating stub modules for COM interfaces and classes.

In COM, type information is compiled from an IDL-file, and stored in a "type library". This is a collection of type information, that is readable (via COM) from erlang.

TBD

1.3 Examples

Detailed examples on how to use Comet

1.3.1 Comet Examples

This chapter describes in detail som examples on Comet usage; the simpler ones first and the most advanced last.

Four examples are given:

- Browsing to a specified address
- Opening Excel, inserting some data, showing a graph
- Calling a function in a C++ library
- Using ADO (ActiveX Data Objects) to read and update data from an SQL Server database.

Source code for these are included in the distribution, in the directory `comet/examples` .

The abbreviations VB and VBA are used for Visual Basic and Visual Basic for Applications.

1.3.2 Requirements

The first example requires that Internet Explorer 4.0 or later is installed.

Example two requires Excel from Office 97 or Office 2000.

The last example can be run as it is, but to modify the COM-library, Visual C++ 5.0 or later is required.

1.3.3 Example One, Opening a Browser to a Specific URL

This example shows how to open a browser (Internet Explorer), and navigate through it to a specific address.

To get the COM interface for the browser, we use a tool such as OLE/COM Object Viewer, which is included in Microsoft's Windows Platform SDK, Visual C and Visual Basic.

Checking the interface for Internet Explorer, we find a couple of things that we need. First, we need the class ID. Then we need the name and parameter list of the funcions and properties required to create and use a browser.

Since starting a browser is not a performance-critical task, we can use the slowest and safest way to do it from Erlang. This means starting the `erl_com` as a port process, and using the IDispatch interface to access Internet Explorer.

Although Internet Explorer provides a dual interface, (that is an interface with both a method table and an IDispatch-interface), the IDispatch interface is safer and slower. Giving it a bad parameter list, returns an error code, rather than a core dump.

To use a COM object, we have to start the server (which starts the port) and start a thread. Then we can create the object, and do what we want with it.

To be able to use constants, we put the source in a module, rather than call it interactively in the Erlang shell.

```
%% an example of using COM with generated code
%%
%% the code was generated with these commands:
%%    erl_com:get_program(a),
%%    erl_com:gen_typelib({a, "c:\program files\microsoft office\office\mso97.dll"}),
%%    erl_com:gen_typelib({a, "c:\program files\microsoft office\office\excel8.olb"}, dispatch),
%%    erl_com:gen_interface({a, "c:\program files\microsoft office\office\excel8.olb"},
%%                          "_Application", dispatch, [{also_prefix_these, ["application"]}, {pref

-module(xc_gen).
-author('jakob@erix.ericsson.se').

-include("erl_com.hrl").
-include("xlChartType.hrl").
-include("xlChartLocation.hrl").
-include("xlRowCol.hrl").

-compile(export_all).

to_cell_col(C) when C > 26 ->
    [C / 26 + 64, C rem 26 + 64];
to_cell_col(C) ->
    [C+64].

populate_area(E, _, _, []) ->
    ok;
populate_area(E, Row, Col, [Data | Resten]) ->
    Cell= to_cell_col(Col)++integer_to_list(Row),
    Range= xapplication:range(E, Cell),
    range:value(Range, Data),
    erl_com:release(Range),
```

```
    populate_area(E, Row+1, Col, Resten).

make_graph(E, Row1, Col1, Row2, Col2, Title) ->
    Charts= xapplication:charts(E),
    NewChart= charts:add(Charts),
    erl_com:release(Charts),
    0= chart:chartType(NewChart, ?xlPieExploded),
    Chart= chart:location(NewChart, ?xlLocationAsObject, "Sheet1"),
    erl_com:release(NewChart),
    R= to_cell_col(Col1)++integer_to_list(Row1)++":"
        ++to_cell_col(Col2)++integer_to_list(Row2),
    Range= xapplication:range(E, R),
    []= chart:setSourceData(Chart, Range, ?xlColumns),
    0= chart:hasTitle(Chart, true),
    ChartTitle= chart:chartTitle(Chart),
    0= chartTitle:caption(ChartTitle, Title),
    erl_com:release(Range),
    erl_com:release(Chart),
    erl_com:release(ChartTitle),
    ok.

sample1() ->
    {ok, _Pid}= erl_com:get_program(xc_gen),
    E= erl_com:create_dispatch(xc_gen, "Excel.Application", ?CLSCTX_LOCAL_SERVER),
    0= xapplication:visible(E, true),
    Wb= xapplication:workbooks(E),
    W= workbooks:add(Wb),
    erl_com:release(W),
    erl_com:release(Wb),
    populate_area(E, 1, 1, ["Erlang", "Java", "C++"]),
    populate_area(E, 1, 2, ["25", "100", "250"]),
    ok= make_graph(E, 1, 1, 3, 2, "Bugs in source code, by language"),
    E.
```

The internet explorer application has a dispatch interface, that implements the IWebBrowser interface. There are a lot of methods. We use the `Navigate` method to open a specific URL, and the `Visible` property to show the browser. (By default, the browser is created invisible, like other Microsoft programs used from COM.)

1.3.4 Example Two, Making a Graph in Excel

In this example, we also start an instance of the Excel application. We use the program name "Excel.Application", which can be used instead of a class ID. This selects the Excel that is installed; Excel from Office 97 or Office 2000.

The easiest way to do anything with Excel is to first record a VBA macro. The resulting VBA macro is shown in figure 1. This macro is manually rewritten a bit to make it simpler. We try it out, and the result is shown in figure 2.

Now, to perform this into Erlang, we have two choices: either we can call the VB code as a subroutine using COM from Erlang, or we can reimplement the VB macro in Erlang. Since this is a user's guide, we of course choose the latter.

To get to the interfaces, we use OLE/COM Object Viewer, and get the IDL for Excel. There is an Excel type library available. We do not want all of it because it is huge. We just pick the needed interfaces, which are _Application, _Graph and _Range. We also extract some enums, which are constants used for parameters in the COM calls.

There are some tricky issues when calling COM from Erlang

First, VB handles releasing of COM interfaces implicitly. Erlang and COM does not do this, so we have to make calls to erl_com:release/1 for every interface we get. For instance, every _Range we get from the property _Application.Range, has to be released. We do this in the helper function data_to_column/3.

Secondly, when an interface is returned, it is returned as an integer. This integer is actually an index into an interface array contained in the erl_com_drv port program. When calling functions in erl_com, we have to provide both the pid and the thread number, so there is a helper function erl_com::package_interface/2, that repackages the interface integer with given thread or other interface. When giving the interface as a parameter to a COM function (through erl_com:call or erl_com:invoke), however, the interface should be converted to a pointer, which is done with the tuple notation for COM types: {vt_unknown, Interface}.

When Excel is started, we execute a series of Excel commands to enter data and to draw a graph. The commands are translated from a VBA macro that we got using Excel's standard macro recorder.

We use some constants that are needed for the Excel commands. These are taken from Visual Basic's code generation from the Excel interfaces. Although these can be fetched from Excel using COM, erl_com does not yet support this. (Future releases will include code-generation that will greatly simplify using big COM-interfaces.

```
-module(xc).
-author('jakob@erix.ericsson.se').

-include("erl_com.hrl").

%% enum XlChartFormat
-define(XlPieExploded, 69).
-define(XlPie, 5).

%% enum XlChartLocation
-define(xlLocationAsNewSheet, 1).
-define(xlLocationAsObject, 2).
-define(xlLocationAutomatic, 3).


%% enum XlRowCol
-define(xlColumns, 2).
-define(xlRows, 1).


-export([populate_area/4, f/3, make_graph/6, sample1/0]).

to_cell_col(C) when C > 26 ->
    [C / 26 + 64, C rem 26 + 64];
to_cell_col(C) ->
    [C+64].

populate_area(E, _, _, []) ->
```

```
    ok;
populate_area(E, Row, Col, [Data | Resten]) ->
    Cell= to_cell_col(Col)++integer_to_list(Row),
    io:format(" ~s ~n ", [Cell]),
    N= erl_com:property_get(E, "range", [Cell]),
    Range= erl_com:package_interface(E, N),
    erl_com:property_put(Range, "Value", Data),
    erl_com:release(Range),
    populate_area(E, Row+1, Col, Resten).

f(E, _, []) ->
    ok;
f(E, Startcell, [Data | Resten]) ->
    {R, C}= Startcell,
    Cell= "R"++integer_to_list(R)++"C"++integer_to_list(C),
    io:format(" ~p ~n ", [Cell]),
    f(E, {R+1, C}, Resten).

make_graph(E, Row1, Col1, Row2, Col2, Title) ->
    Charts = erl_com:package_interface(E, erl_com:property_get(E, "Charts")),
    erl_com:invoke(Charts, "Add"),
    ActiveChart= erl_com:package_interface(E, erl_com:property_get(E, "ActiveChart")),
    erl_com:property_put(ActiveChart, "ChartType", {vt_i4, ?XlPieExploded}),
    erl_com:invoke(ActiveChart, "Location", [{vt_i4, ?xlLocationAsObject}, "Sheet1"]),
    Chart= erl_com:package_interface(E, erl_com:property_get(E, "ActiveChart")),
    R= to_cell_col(Col1)++integer_to_list(Row1)++":"
        ++to_cell_col(Col2)++integer_to_list(Row2),
    io:format(" ~s ~n ", [R]),
    Range= erl_com:property_get(E, "Range", [R]),
    erl_com:invoke(Chart, "SetSourceData", [{vt_unknown, Range}, {vt_i4, ?xlColumns}]),
    erl_com:property_put(Chart, "HasTitle", true),
    ChartTitle= erl_com:package_interface(E, erl_com:property_get(Chart, "ChartTitle")),
    erl_com:property_put(ChartTitle, "Caption", Title).
                                            %erl_com:release(erl_com:package_interface(E, Range)),
                                            %erl_com:release(ActiveChart),
                                            %erl_com:release(Charts).

sample1() ->
    {ok, Pid}= erl_com:start_process(),
    T= erl_com:new_thread(Pid),
    E= erl_com:create_dispatch(T, "Excel.Application", ?CLSCTX_LOCAL_SERVER),
    erl_com:property_put(E, "Visible", true),
    Wb= erl_com:package_interface(T, erl_com:property_get(E, "Workbooks")),
    erl_com:invoke(Wb, "Add"),
    populate_area(E, 1, 1, ["Erlang", "Java", "C++"]),
    populate_area(E, 1, 2, ["25", "100", "250"]),
    make_graph(E, 1, 1, 3, 2, "Programfel i Ericssonprojekt, sprkuppdelning"),
    {T, E, Wb}.
```

Now, from version 1.1 of comet, there is a possibility to generate code stubs that wrapps the `erl_com` calls in shorter and clearer names. If we use `erl_com:gen_typelib(X, dispatch)` to generate files, where X is an interface for an Excel object, we have a more readable form for the above:

```
%% an example of using COM with generated code
%%
%% the code was generated with these commands:
%%    erl_com:get_program(a),
%%    erl_com:gen_typelib({a, "c:\program files\microsoft office\office\mso97.dll"}),
%%    erl_com:gen_typelib({a, "c:\program files\microsoft office\office\excel8.olb"}, dispatch),
%%    erl_com:gen_interface({a, "c:\program files\microsoft office\office\excel8.olb"},
%%                          "_Application", dispatch, [{also_prefix_these, ["application"]}, {pref

-module(xc_gen).
-author('jakob@erix.ericsson.se').

-include("erl_com.hrl").
-include("xlChartType.hrl").
-include("xlChartLocation.hrl").
-include("xlRowCol.hrl").

-compile(export_all).

to_cell_col(C) when C > 26 ->
    [C / 26 + 64, C rem 26 + 64];
to_cell_col(C) ->
    [C+64].

populate_area(E, _, _, []) ->
    ok;
populate_area(E, Row, Col, [Data | Resten]) ->
    Cell= to_cell_col(Col)++integer_to_list(Row),
    Range= xapplication:range(E, Cell),
    range:value(Range, Data),
    erl_com:release(Range),
    populate_area(E, Row+1, Col, Resten).

make_graph(E, Row1, Col1, Row2, Col2, Title) ->
    Charts= xapplication:charts(E),
    NewChart= charts:add(Charts),
    erl_com:release(Charts),
    0= chart:chartType(NewChart, ?xlPieExploded),
    Chart= chart:location(NewChart, ?xlLocationAsObject, "Sheet1"),
    erl_com:release(NewChart),
    R= to_cell_col(Col1)++integer_to_list(Row1)++":"
        ++to_cell_col(Col2)++integer_to_list(Row2),
    Range= xapplication:range(E, R),
    []= chart:setSourceData(Chart, Range, ?xlColumns),
    0= chart:hasTitle(Chart, true),
    ChartTitle= chart:chartTitle(Chart),
    0= chartTitle:caption(ChartTitle, Title),
    erl_com:release(Range),
    erl_com:release(Chart),
    erl_com:release(ChartTitle),
    ok.

sample1() ->
```

```
    {ok, _Pid}= erl_com:get_program(xc_gen),
    E= erl_com:create_dispatch(xc_gen, "Excel.Application", ?CLSCTX_LOCAL_SERVER),
    0= xapplication:visible(E, true),
    Wb= xapplication:workbooks(E),
    W= workbooks:add(Wb),
    erl_com:release(W),
    erl_com:release(Wb),
    populate_area(E, 1, 1, ["Erlang", "Java", "C++"]),
    populate_area(E, 1, 2, ["25", "100", "250"]),
    ok= make_graph(E, 1, 1, 3, 2, "Bugs in source code, by language"),
    E.
```

To use the code above, we have to generate Erlang stub modules for a lot of interfaces. Checking with the OLE/COM viewer, we see that Excel uses both it's own type library, and a common library for office programs. We generate these. We use an object `application` a lot, however that name is used by another module in OTP. So we generate specifically the "_Application" interface, with a prefix "x", to easily use it. Now the interface for "_Application" is in the module `xapplication`.

When we use the functions, we take care to match every call. This is to catch errors early, remember erl_com returns errors in a {`com_error, ...`} tuple. Successful `invoke` and `property_put` returns `[]` and `0`, respectively.

1.3.5 Example three, calling a COM object in C++

To be done.

1.3.6 Example four, using ActiveX Data Objects from Erlang

ActiveX Data Objects, or ADO for short, is Microsoft's new components for data access, using either Ole-DB or ODBC. They provide a nice COM wrapper for accessing SQL databases such as SQL Server, Oracle and Sybase.

The following code snippets uses generated code to access data on SQL server, in the example database "PUBS". For information on ADO, refer to Microsoft documentation.

```
-module(ado).
-author('jakob@erix.ericsson.se').

-compile(export_all).
%%-export([Function/Arity, ...]).

%% these are generated from ADO:
%% erl_com:create_object(Ado, "ADODB.Connection"), com_gen:gen_typelib(Ado).

-include("cursorlocationenum.hrl").
-include("cursortypeenum.hrl").
-include("locktypeenum.hrl").
-include("commandtypeenum.hrl").

select_sample() ->
    Sql= "select * from titles order by title",
    select_sample(Sql).
```

```
select_sample(Sql) ->
    {ok, Pid1} = erl_com:get_program(a),
    C= connection_class:create_object(a),
    %% Load the Driver and connect to the database.
    Strconn= "Provider=SQLOLEDB;Initial Catalog=pubs;"
        "Data Source=eomer;User Id=sa;Password=;",
    connection:open(C, Strconn),
    %% do the select
    Rs= connection:execute(C, Sql),
    %% get Fields
    Fields= recordset:fields(Rs),
    N= fields:count(Fields),
    %% get names
    Fl= lists:map(fun(J) -> fields:item(Fields, J) end, lists:seq(0, N-1)),
    %% get each field
    Nl= lists:map(fun(F) -> field:name(F) end, Fl),
    %% read values
    Vl= read_all(Rs, Fl, recordset:eOF(Rs), [Nl]),
    erl_com:release(Fields),
    erl_com:release(Rs),
    erl_com:release(C),
    Vl.


read_row(Fl) ->
    lists:map(fun(F) -> field:value(F) end, Fl).

%% read all values
read_all(Rs, Fl, true, Acc) ->
    lists:reverse(Acc);
read_all(Rs, Fl, false, Acc0) ->
    Acc= [read_row(Fl) | Acc0],
    recordset:moveNext(Rs),
    %% limit to 100 records
    read_all(Rs, Fl, (length(Acc) > 100) or recordset:eOF(Rs), Acc).


map2_(F, [], _, Acc) ->
    Acc;
map2_(F, _, [], Acc) ->
    Acc;
map2_(F, [A0 | A], [B0 | B], Acc0) ->
    Acc= [F(A0, B0) | Acc0],
    map2_(F, A, B, Acc).

map2(F, A, B) ->
    lists:reverse(map2_(F, A, B, [])).

map3_(F, [], _, _, Acc) ->
    Acc;
map3_(F, _, [], _, Acc) ->
    Acc;
map3_(F, _, _, [], Acc) ->
```

```
    Acc;
map3_(F, [A0 | A], [B0 | B], [C0 | C], Acc0) ->
    Acc= [F(A0, B0, C0) | Acc0],
    map3_(F, A, B, C, Acc).

map3(F, A, B, C) ->
    lists:reverse(map3_(F, A, B, C, [])).


insert_sample() ->
    %% Start a new COM server. The application must already be started.
    {ok, Pid1} = erl_com:get_program(a),
    %% Load the Driver and connect to the database, make recordset directly
    Strconn= "Provider=SQLOLEDB;Initial Catalog=pubs;Data Source=eomer;User Id=sa;Password=;",
    Strsql= "select * from titles",
    Rs= recordset_class:create_object(a),
    recordset:open(Rs, Strsql, Strconn, ?adOpenForwardOnly, ?adLockOptimistic),
    %% Add a new row
    recordset:addNew(Rs),
    Fields= recordset:fields(Rs),
    N= fields:count(Fields),
    %% Get each field
    Fl= lists:map(fun(J) -> fields:item(Fields, J) end, lists:seq(0, N-1)),
    Nl= lists:map(fun(F) -> field:name(F) end, Fl),
    %% Fields: title_id, title, type, pub_id, price, advance, royalty, ytd_sales, notes, pubdate
    %% Have some nice values
    FVals = ["TC8789", "Kul med prolog?", "UNDECIDED   ", "1389", 8.99,
             8000, 10, 2000, "Det r inte S kul med Prolog.", 0],
    %% Set values of new row
    map3(fun(F, V, Na) ->
                 io:format("name ~s value ~p ~n", [Na, V]),
                 []= field:value(F, V) end, Fl, FVals, Nl),
    %% "Commit" to the DB
    recordset:update(Rs).

delete_sample(Title_id) ->
    {ok, Pid1} = erl_com:get_program(a),
    %% Load the Driver and connect to the database, create recordset directly
    Strconn= "Provider=SQLOLEDB;Initial Catalog=pubs;Data Source=eomer;User Id=sa;Password=;",
    Strsql= "select * from titles",
    Filter= "title_id='" ++ Title_id ++ "'",
    Rs= recordset_class:create_object(a),
    []= recordset:open(Rs, Strsql, Strconn, ?adOpenForwardOnly, ?adLockOptimistic),
    %% Set the filter, required for delete (I think?)
    []= recordset:filter(Rs, Filter),
    %% Delete
    []= recordset:delete(Rs),
    %% And "commit"
    []= recordset:update(Rs).
```

ADO is a nice way of accessing databases from a windows platform. The example shows both reading and changing data on the database. The code is more or less taken from Microsoft's documentation on ADO.

1.4 Comet Release Notes

Notes for Comet

1.4.1 Comet 1.1.1

Improvements and New Features

- Small improvements on wrapper code generation.
- Support for enumeration interfaces.

Known Bugs and Problems

No way to avoid context switches (in Win32) when calling COM-object. This might be addressed in future versions.

Some types from code generations are wrong, and needs manual adjustment (i.e. Word 2000).

1.4.2 Comet 1.1

New release.

Greatly improved everything.

Comet can now be run as a port program and/or a port driver. The same API works in both.

Now generates Erlang wrapper modules around COM objects and interfaces, using information in COM TypeLibs.

Examples include controlling Internet Explorer, using Excel and accessing data through ADO.

1.4.3 Comet 1.0

New application.

Comet Reference Manual

Short Summaries

- Application comet [page 19] – The COM client for Erlang
- Erlang Module com_gen [page 20] – Comet code generator from COM type libraries.
- Erlang Module erl_com [page 23] – Comet gen_server with API to Erlang COM client.

comet

No functions are exported.

com_gen

The following functions are exported:

- `gen_enum(ComInterface, EnumName) -> {Erlfilename, Hrlfilename, ok}`
  [page 21] Generate an enum to a module and a header.
- `gen_enum(ComInterface, EnumName, Options) -> {Erlfilename, Hrlfilename, ok}`
  [page 21] Generate an enum to a module and a header.
- `gen_coclass(ClassName, ClsID) -> {ok, Filename}`
  [page 21] Generate a file for a COM class.
- `gen_coclass(ClassName, ClsID, Options) -> {ok, Filename}`
  [page 22] Generate a file for a COM class.
- `gen_interface(ComInterface, VirtualOrDispatch) -> {ok, Filename}`
  [page 22] Start or get a named server.
- `gen_interface(ComInterface, VirtualOrDispatch, Options) -> {ok, Filename}`
  [page 22] Start or get a named server.
- `gen_interface(ComInterface, IntfName, VirtualOrDispatch) -> {ok, Filename}`
  [page 22] Start or get a named server.
- `gen_interface(ComInterface, IntfName, VirtualOrDispatch, Options) -> {ok, Filename}`
  [page 22] Start or get a named server.

erl_com

The following functions are exported:

- `start_program() -> {ok, Pid}`
  [page 24] Start the server and start a port program (in a separate process).
- `start_program(ServerName) -> {ok, Pid}`
  [page 24] Start the server and start a port program (in a separate process).
- `get_program(ServerName) -> {ok, Pid}`
  [page 24] Start the server and start a port program (in a separate process).
- `start_driver() -> {ok, Pid}`
  [page 24] Start the server and load a port driver.
- `start_driver(ServerName) -> {ok, Pid}`
  [page 24] Start the server and load a port driver.
- `get_driver(ServerName) -> {ok, Pid}`
  [page 24] Start the server and load a port driver.
- `get_or_start(Name, ProgramFlag) -> {ok, Pid}`
  [page 25] Start or get a named server.
- `stop(ServerRef) -> ok`
  [page 25] Stop the server.
- `new_thread(ServerRef | PrevComThread) -> ComThread`
  [page 25] Create a new Windows thread for COM execution.
- `end_thread(ComThread) -> ok`
  [page 25] End a Windows thread in Comet.
- `create_object(ThreadOrServer, Class) -> ComInterface`
  [page 26] Create a COM object.
- `create_object(ThreadOrServer, Class, Ctx) -> ComInterface`
  [page 26] Create a COM object.
- `create_object(ThreadOrServer, Class, RefID) -> ComInterface`
  [page 26] Create a COM object.
- `create_object(ThreadOrServer, Class, RefID, Ctx) -> ComInterface`
  [page 26] Create a COM object.
- `create_dispatch(ThreadOrServer, Class) -> ComInterface`
  [page 26] Create a COM object.
- `create_dispatch(ThreadOrServer, Class, Ctx) -> ComInterface`
  [page 26] Create a COM object.
- `get_object(ThreadOrServer, Name) -> ComInterface`
  [page 26] Get a COM object.
- `get_object(ThreadOrServer, Name, Interface) -> ComInterface`
  [page 26] Get a COM object.
- `get_dispatch(ThreadOrServer, Name) -> ComInterface`
  [page 26] Get a COM object.
- `query_interface(ComInterface, Iid)`
  [page 27] Get a COM interface from another.
- `release(ComInterface)`
  [page 27] Release a COM interface.

- com_call(ComInterface, MethodOffs)
  [page 27] Call a COM method, using the V-table.
- com_call(ComInterface, MethodOffs, Pars)
  [page 27] Call a COM method, using the V-table.
- invoke(ComInterface, MethodName, Pars)
  [page 28] Invoke a COM method, using the dispatch interface.
- invoke(ComInterface, MethodID, Pars)
  [page 28] Invoke a COM method, using the dispatch interface.
- property_get(ComInterface, MethodID)
  [page 28] Get a COM property, using the dispatch interface.
- property_get(ComInterface, MethodID, [Parameters])
  [page 28] Get a COM property, using the dispatch interface.
- property_get(ComInterface, MethodName)
  [page 28] Get a COM property, using the dispatch interface.
- property_get(ComInterface, MethodName, [Parameters])
  [page 28] Get a COM property, using the dispatch interface.
- property_put(ComInterface, MethodName, Value)
  [page 28] Set a COM property, using the dispatch interface.
- property_put(ComInterface, MethodName, [Parameters], Value)
  [page 28] Set a COM property, using the dispatch interface.
- property_put(ComInterface, MethodID, Value)
  [page 28] Set a COM property, using the dispatch interface.
- property_put(ComInterface, MethodID, [Parameters], Value)
  [page 28] Set a COM property, using the dispatch interface.
- property_put_ref(ComInterface, MethodName, Value)
  [page 28] Set a COM ref property, using the dispatch interface.
- property_put_ref(ComInterface, MethodName, [Parameters], Value)
  [page 28] Set a COM ref property, using the dispatch interface.
- property_put_ref(ComInterface, MethodID, Value)
  [page 28] Set a COM ref property, using the dispatch interface.
- property_put_ref(ComInterface, MethodID, [Parameters], Value)
  [page 28] Set a COM ref property, using the dispatch interface.
- package_interface(ThreadOrInterface, NewIntfNum) -> NewComInterface
  [page 28] Convert an integer return from a COM call, to an interface tuple.
- get_method_id(DispatchInterface, MethodName) -> MethodID
  [page 28] Convert a named method to its corresponding ID
- get_interface_info(ComInterface, VirtualOrDispatch) -> TypeInfo
  [page 29] Return interface information from the COM TypeLib.
- get_interface_info(ComInterface, TypeName, VirtualOrDispatch) -> TypeInfo
  [page 29] Return interface information from the COM TypeLib.
- get_typelib_info(ComInterface) -> TypeLibInfo
  [page 30] Return information on types in a COM type library.
- test(ComInterface) -> []
  [page 30] Test function, break into Windows debugger.
- enum(ComInterface) -> ComEnum
  [page 30] Get an iterator (called enum in COM) for a COM-object.

- `next(ComEnum) -> Variant`
  [page 30] Get the next item of a COM iterator.
- `nexti(ComEnum) -> ComInterface`
  [page 30] Get the next item of a COM iterator.
- `intfenum_next(ComEnum) -> ComInterface`
  [page 30] Get the next item of a COM iterator.
- `map_enum(ComEnum, Fun)`
  [page 30] Map over an COM iterator.
- `map_enumi(ComEnum, IFun)`
  [page 30] Map over an COM iterator.
- `map_intfenumi(ComEnum, IFun)`
  [page 30] Map over an COM iterator.

comet

Application

Comet, the COM client for Erlang, is a way to call any COM-service in Windows from an Erlang program. It is a combination of a gen_server and a port program (or port driver) that enables Erlang programs to call almost any COM-server.

Comet uses a gen_server in the module erl_com, together with a port program (or port driver), to call COM-servers. Both the late-binding interface IDispatch and early-binding virtual interfaces can be used. Erlang types are converted to COM types and parameters are returned.

COM stands for Component Object Model, (or sometimes Common Object Model), and is MicroSoft's technique for component-based programming. It allows programs on Windows systems, to call other programs and libraries across language boundaries. It is a competitor to CORBA, but has other functionality too.

COM is available on all 32-bit Windows systems, such as NT 4, Windows 95, Windows 98 and Windows 2000. Comet can be used on any of these.

With Comet, an Erlang application can use (almost) any COM-service on Windows from Erlang. Examples of what can be done include:

- Opening webpages with Internet Exporer (or Netscape)
- Reading and writing data from Excel Worksheet
- Reading and writing from Word
- Call C-code-libraries an efficient way, without the hassle of creating a port-driver
- Execute scripts in VBScript or JavaScript

Each of these examples are described in `comet_examples`.

Using Comet for other purposes than those described in the examples, requires knowledge of COM. COM is a complex topic, and there are a lot of books written about it. Three decent references are given below. Some sections of the Comet documentation assumes basic knowledge of COM.

Configuraion

Comet does not use any configuration parameters.

See Also

erl_com(3), comet_examples(3), some decent books on COM, such as: Inside COM, by Dale Rogerson (ISBN: 1572313498) Essential COM, by Don Box (ISBN: 0201634465) Mastering COM and COM+, by Rofail and Shohoud (ISBN: 0782123848)

com_gen

Erlang Module

The com_gen module generates stub code from COM type libraries. This makes it as easy to use COM objects in erlang as in other languages with COM capabilies (such as Visual Basic or Microsoft Java). OK, not quite as easy, but not really difficult anyway.

It also makes it possible to use the early-binding virtual interfaces available for most COM classes. This is often way faster than going through the Dispatch/Invoke-interface.

The com_gen module takes a COM interface and generates erlang source code files in the current directory. It can generate just one type, or a whole typelib with several types. There are options controlling the output.

These options are available, a (D) indicates a default option:

```
Option           Explanation
=================================================================
fix_names (D)    Fix names to reasonable Erlang names

keep_names       Keeps the names as is, requiring '

prefix           A prefix added before each name

prefix_these     A list of names to prefix (empty means all)

also_prefix_these
                 Include the given names to be prefixed

dont_prefix_these
                 Remove the given names from the list of
                 names being prefixed

suffix           A suffix added after each name

class_suffix     A suffix added after class names

{verbose, Lvl}   The amount of feedback while generating
      1 (D)      Only write type library name
      2          Write each type in a library
      3          Write each member in a type (e.g. each function)

no_optional      Prevents generation of multiple functions with
                 optional arguments

virtual          Generate virtual interfaces (dispatch interfaces are
                 generated with the suffix _disp)
```

```
no_base_virtual  Generate virtual interfaces without including
                 functions in the interhited interface in each
                 interface module
```

The name fixing is made to make it easier to use COM object in erlang. It removes any initial _ (underscores), and changes the first letter to lower case. Thus _Connection -> connection and EOF -> eOF. Without the name fixing (option `keep_names`), each COM name has to be surrounded with ' (ticks).

Sometimes names are unappropriate or clashes with other names, e.g. _Application -> application. In those cases a prefix can be given.

By default, the following names will be prefixed with `c_`: `end`, `fun`, `when`. Names that might be added are `application`, `receive`, `throw`, etc.

Each interface generates a file with stubs that calls the appropriate `erl_com` function. If the function has optional in arguments, several version with different cardinality is generated.

Virtual interfaces that inherits from other interfaces, are normally recursively generated to include there base interface functions. This makes comet behave more like C++ or Visual Basic when using interfaces with inheritance. (This can be turned off with the `no_base_virtual` option.)

When generating from a library, lot's of files will be created. All types in the library that comet recognizes will be generated as files.

Note that the options for virtual or dispatch does not prevent dispatch-interfaces to be generated, however they will have another name.

Often in COM, the same name is used for different members. To avoid name clashes, `com_gen` has a simple rule: properties are prefixed with `put_`, `get_` or `put_ref_`, and methods are suffixed with _ (underscore). Name changes will be reported with warnings from `com_gen`.

The current default options for `com_gen` are: use name fixing, prefix `c_`, no suffix, class suffix `_class`, verbose level 1 (print type names, but not mebmers), generate multiple functions for optional in parameters and prefer dispatch-interfaces.

Exports

```
gen_enum(ComInterface, EnumName) -> {Erlfilename, Hrlfilename, ok}
gen_enum(ComInterface, EnumName, Options) -> {Erlfilename, Hrlfilename, ok}
```

Types:

- EnumName = string()

The `gen_enum` function is used to generate a COM enum from a type library. The enum yields both a header file and a module with the same contents.

The header file contains the enum values as defines, for inclusion in an erlang module.

The module file contains the values as functions, and is useful when experimenting in an erlang shell.

```
gen_coclass(ClassName, ClsID) -> {ok, Filename}
```

`gen_coclass(ClassName, ClsID, Options) -> {ok, Filename}`

Types:

- ClassName = ClsID = Filename = string()

This functions generates a stub for a COM class. Both the name and the GUID has to be specified.

The erlang module only contatins create-functions for instance creation, and `iid` and `name`.

Note that no checking is performed on the `ClsID` parameter. This function is rarely used, instead generation is done from the type library with `gen_typelib`.

`gen_interface(ComInterface, VirtualOrDispatch) -> {ok, Filename}`
`gen_interface(ComInterface, VirtualOrDispatch, Options) -> {ok, Filename}`
`gen_interface(ComInterface, IntfName, VirtualOrDispatch) -> {ok, Filename}`
`gen_interface(ComInterface, IntfName, VirtualOrDispatch, Options) -> {ok, Filename}`

Types:

- VirtualOrDispatch = virtual | dispatch
- IntfName, Filename = string()

This function generates stub files for an interface. For each of the methods (and property functions) in the interface, a function is generated.

The `VirtualOrDispatch` flag is used to specify if virtual interfaces are to be generated. If it `virtual` is given, virtual and dispatch versions are generated, the dispatch interfaces have the suffix _disp. If `dispatch` is given, only dispatch interfaces are generated, without suffix.

The options can specify whether several versions for optional parameters is generated, and if inherited functions should be included in virtual interfaces.

Note that currently, no optional out parameters provided in the interface stubs, to use them, `erl_com:invoke` has to be called directly.

erl_com

Erlang Module

The erl_com module is a gen_server that exposes an API to the port program and port driver that is used to call COM services from Erlang.

There is a mapping between types in Erlang and types in COM. The following table shows how Erlang types are converted by the port program to COM types.

COM type	Erlang type	Comment
VT_I4	integer()	
VT_U4	integer()	
VT_BOOL	true | false	
VT_BSTR	string()	Strings are translated between Ascii and Unicode
VT_DATE	{integer(), integer(), integer()}	Same format as returned from now()
	{{Year, Month, Day}, {Hour, Min, Sec}}	Date and time, with integers in tuples
VT_PTR	{vt_*, out}	Any output parameter, including return value
VT_I1	{vt_i1, integer()}	
VT_U1	{vt_u1, integer()}	
VT_I2	{vt_i2, integer()}	
VT_U2	{vt_u2, integer()}	
VT_R8	float()	
VT_R4	{vt_r4, float()}	
VT_CY	{vt_cy, float()}	Note that the precision is lower on float()
VT_DECIMAL	{vt_decimal, float()}	-"-
VT_UNKNOWN	integer()	Should be sent to package_interface
VT_DISPATCH	integer()	-"-
other types	unsupported	

Some of the internal Erlang types map to types in COM. Most types in COM, however, have no corresponding type in Erlang. In these cases, a special tuple is used, of the form {ComType, Value}, where ComType is the corresponding type-name as defined in ole2.h in the Microsoft Windows SDK.

In the functions below, the ComInterface is used. It is a tagged tuple, that identifies a COM interface in the port driver.

```
ComInterface = {com_interface, pid(), ThreadNo, InterfaceNo}
ThreadNo = InterfaceNo = integer()
```

Exports

```
start_program() -> {ok, Pid}
start_program(ServerName) -> {ok, Pid}
get_program(ServerName) -> {ok, Pid}
```

Types:

- Pid = pid()
- ServerName = atom()

Starts a new server, and initializes the COM port. Also starts one thread for running COM calls.

This function starts the COM port as a port-program, in a separate process. The erl_com gen_server uses (as usual in Erlang), a pipe to communicate with the port. This has the benifit that a crash in the COM port, will not crash the emulator.

Each `erl_com` server starts a separate port-program.

The server can be started with or without a registered name.

Normally, only one `erl_com` server is started on a node, using the `get_program/1` call, with possibly several threads for several clients. The only reason to start more than one server on the same node is if one crashes, then the others will keep on running.

This way to launch Comet can be used when:

- The COM server is not 100% certain and crashproof.
- The overhead of using a separate port process is acceptable.

Since this way is safer, it is the preferred way of using comet.

```
start_driver() -> {ok, Pid}
start_driver(ServerName) -> {ok, Pid}
get_driver(ServerName) -> {ok, Pid}
```

Types:

- Pid = pid()
- ServerName = atom

Starts a new server, and initializes the COM port. Also starts one thread for running COM calls.

The port is loaded as a port driver. This is the most efficient way to use COM, since the com port resides in the same process as the Erlang emulator. However this also means that crashing COM-objects will bring down the emulator.

The server can be started with or without a registered name. There is no advantage of having two servers on the same node.

The `get_driver/1` call, gets a named process, or starts one if no one is running.

This way to launch Comet should only be used in two situations:

- When the COM servers are in separate processes, where they will not bring down the emulator in case of a crash.
- When the COM server is well-known and unlikely to crash, and the overhead of using a separate port process is unacceptable.

`get_or_start(Name, ProgramFlag) -> {ok, Pid}`

Types:

- Name = atom()
- ProgramFlag = program | driver

Calls `get_program` or `get_driver`, depening on the `ProgramFlag` parameter.

`stop(ServerRef) -> ok`

Types:

- ServerRef = Name | Pid
- Name = atom()
- Node = atom()
- Pid = pid()
- Thread = integer()
- Error = {com_error, Errcode}
- Errcode = string()

Shuts the `erl_com` server down. This will stop any threads. Interfaces should be released before.

(Remember COM has no garbage collection!)

`new_thread(ServerRef | PrevComThread) -> ComThread`

Types:

- ServerRef = Name | Pid
- PrevComThread = ComThread = {com_thread, ServerRef, ThreadNo}
- Thread = integer()

Creates a new Windows thread that can be used to create and manipulate COM objects. This is done automatically after `erl_com` is started. One thread is created.

To allow COM calls to take time without blocking the emulator, `erl_com` allows multi-threaded execution. The maximum number of threads is 60. However, creating more than a few is not useful for practical reasons.

When a COM-thread is created, it is suspended with a select function (which is called WaitForMultipleObjects in the Win32 API). Calling any COM-functions from the thread, is done by setting up a parameter buffer and signaling an event, that wakes up the thread.

The return value is a tuple that includes `Thread`, a thread index that is an integer between 0 and 60, which is unique for each thread, and allocated incrementally. Thread index values will be reused if a thread is ended.

All COM calls are asynchronous from the emulators view, they are never called from the emulator main thread, and thus only blocks the calling Erlang process.

`end_thread(ComThread) -> ok`

Types:

- ComThread = {com_thread, ServerRef, ThreadNo}
- ThreadNo = integer()

Ends a thread previously created with `new_thread`. If the thread has any interfaces, these must be released before the thread is ended, otherwise resource leakage can occur. (Remember COM has no garbage collection!)

The thread is signaled and will exit. The thread index will be marked as available, internally in the port program.

```
create_object(ThreadOrServer, Class) -> ComInterface
create_object(ThreadOrServer, Class, Ctx) -> ComInterface
create_object(ThreadOrServer, Class, RefID) -> ComInterface
create_object(ThreadOrServer, Class, RefID, Ctx) -> ComInterface
create_dispatch(ThreadOrServer, Class) -> ComInterface
create_dispatch(ThreadOrServer, Class, Ctx) -> ComInterface
```

Types:

- ThreadOrServer = ComThread | ServerRef
- ServerRef = Pid | Name
- Pid = pid()
- Name = atom()
- ComThread = {com_thread, Pid, ThreadNo}
- Class = string()
- RefID = string()
- Ctx = integer()
- ThreadNo = integer()
- InterfaceNum = integer()

This function creates a COM object. It calls the Win32 API function, `CoCreateInstance`. Refer to Windows documentation. The string `Class` can be either a GUID for a class, or a COM program string. Values for the `Ctx` are defined in `erl_com.hrl`. If no `Ctx` is given, all flags are set to one (using any local service).

When successful, this function creates a COM object, and returns a tuple `ComInterface`, which is a handle for the object, that is used for calling methods, and releasing the object.

The `create_dispatch` variant creates an object with the `IDispatch` interface. The interface wanted can be specified in the `RefID` parameters.

```
get_object(ThreadOrServer, Name) -> ComInterface
get_object(ThreadOrServer, Name, Interface) -> ComInterface
get_dispatch(ThreadOrServer, Name) -> ComInterface
```

Types:

- ThreadOrServer = ComThread | ServerRef
- ServerRef = Pid | Name
- Pid = pid()
- Name = atom()
- ComThread = {com_thread, Pid, ThreadNo}

- Name = string()
- Interface = string()

This function gets a COM object. It calls the Win32 API function, `CoGetObject`. Refer to Windows documentation. The string `name` is a name that is used to get the object using a moniker. The `bindOptions` parameter of `CoGetObject` always contains default values.

When successful, this function references a COM object, and returns a tuple `ComInterface`, which is a handle for the object, that is used for calling methods, and releasing the object.

The `get_dispatch` variant gets an object with the `IDispatch` interface. Other interface wanted can be specified in the `Interface` parameter.

`query_interface(ComInterface, Iid)`

Types:

- Iid = string()

Calls `query_interface` on the given interface. Note that in COM, an object is also considered an interface.

This function is used to see what interfaces an object implements and to do down-casting.

`release(ComInterface)`

In COM, all interfaces are reference-counted. The `release` function decrements the reference counter, and releases the interface (or object) if it reaches zero. Note that it is important to release all objects created, and interfaces acquired. Otherwise resource leaking will occur. Future versions of `comet` may provide for GC of COM objects.

This function in `erl_com` also returns the `ComInterface` tuple, after `release` it is not allowed to use the `ComInterface`.

`com_call(ComInterface, MethodOffs)`
`com_call(ComInterface, MethodOffs, Pars)`

Types:

- MethodOffs = integer()
- Pars = list()

This is the way to call a method in a virtual COM interface. Beware that the parameter types must match the types in the COM interface function. Any type errors, or bad parameter counts, will crash the COM driver.

Note that return values are handled with `out` parameters when using `com_call/3`. (As opposed to `invoke/3`).

This function should not be called explicitly, only from generated code (see `com_gen`).

Exports

`invoke(ComInterface, MethodName, Pars)`
`invoke(ComInterface, MethodID, Pars)`

Types:

- MethodName = string()
- MethodID = integer()

There are two ways to call a method in a COM interface. A dispatch-interface, has a method invoke, that is used to call methods. This method is intended for interpreted languages. The invoke method is safer than `com_call`, but also slower.

In many cases, the overhead of using invoke, is not significant. Therefore, it should be preferred, since it has parameter checking, better error messages, etc.

The return vaule sometimes needs a bit of processing. In particular, an interface is returned as an integer only, and the function `package_interface` must be called (see below).

`property_get(ComInterface, MethodID)`
`property_get(ComInterface, MethodID, [Parameters])`
`property_get(ComInterface, MethodName)`
`property_get(ComInterface, MethodName, [Parameters])`

To get a property value through the dispatch-interface, this function is used.

`property_put(ComInterface, MethodName, Value)`
`property_put(ComInterface, MethodName, [Parameters], Value)`
`property_put(ComInterface, MethodID, Value)`
`property_put(ComInterface, MethodID, [Parameters], Value)`

To set a property value through the dispatch-interface, this function is used.

`property_put_ref(ComInterface, MethodName, Value)`
`property_put_ref(ComInterface, MethodName, [Parameters], Value)`
`property_put_ref(ComInterface, MethodID, Value)`
`property_put_ref(ComInterface, MethodID, [Parameters], Value)`

To set a property reference through the dispatch-interface, this function is used.

`package_interface(ThreadOrInterface, NewIntfNum) -> NewComInterface`

Types:

- ThreadOrInterface = ComThread | ComInterface

This function converts an interface number, as returned from `erl_com` when interface-returning COM calls are made, into an interface tuple. This interface tuple can be used in other COM calls.

Note that this function is called in generated code (see `com_gen`).

`get_method_id(DispatchInterface, MethodName) -> MethodID`

Types:

- DispatchInterface = ComInterface
- MethodName = string()
- MethodID = integer()

Finds the ID of a method (or property), given its name. The interface must be a dispatch-interface.

```
get_interface_info(ComInterface, VirtualOrDispatch) -> TypeInfo
get_interface_info(ComInterface, TypeName, VirtualOrDispatch) -> TypeInfo
```

Types:

- VirtualOrDispatch = virtual | dispatch
- TypeName = string()
- TypeInfo = EnumInfo | InterfaceInfo | ClassInfo
- EnumInfo = {enum, virtual, TypeId, [EnumMember...], [Subtype...]}
- TypeId = {Name, IID}
- Name = IID = string()
- EnumMember = {EnumName, EnumValue}
- EnumValue = integer()
- ClassInfo = {coclass, virtual, TypeId, [], []}
- InterfaceInfo = DispatchInfo | VirtualInfo
- DispatchInfo = {dispatch, IntfKind, TypeId, [Func...], [Subtype...]}
- VirtualInfo = {interface, IntfKind, TypeId, [Func...], [Subtype...]}
- IntfKind = dual | dispatch | virtual
- Func = {FuncName, [InvKind], FuncType, IdOrOffset, [Parameter...], ReturnValue}
- EnumName = FuncName = string()
- InvKind = func | property_get | property_put | property_put_ref
- FuncType = virtual | purevirtual | nonvirtual | static | dispatch
- IdOrOffset = integer
- ReturnValue = ComType | void
- Parameter = {ParamName, [ParamFlag...], ComType, DefaultValue
- ParamName = string()
- ParamFlag = in | out | lcid | retval | optional | has_default | has_custom_data
- ComType = vt_i4 | vt_str | ... see above
- DefaultValue = {ComType, Value} | {}
- SubType = TypeId

How about that? If it looks complicated it's because it is.

The `get_interface_info` is used to retrieve information from a COM type library. It is actually a misnomer, it's not just for interfaces, but also for enums and coclasses. Other types of types are unsupported by comet (currently).

Given an interface and a type name, it fetches most of the information available in the typelibrary, using the `ITypeInfo` and `ITypeLib` interfaces. It is kind of an erlang version of the OLE/COM object viewer in the Windows SDK. An interface can be listed as a dispatch or a virtual interface.

This function is used by `com_gen` to provide erlang stub generation from type libraries.

To understand its output, refer to the COM documentation on ITypeInfo and ITypeLib, or to a book.

There is currently no way in comet to retrieve information from a type-library without creating at least one object from it. This might be improved in later releases.

`get_typelib_info(ComInterface) -> TypeLibInfo`

Types:

- TypeLibInfo = {TypeLibName, [TypeInfo...]}
- TypeInfo = {TypeKind, TypeName, IID}
- TypeKind = enum | record | module | interface | dispatch | coclass | alias | union

The `get_typelib_info` function lists all types in a COM type library. It is used by `com_gen` to generate stub code.

Note that only enums, interfaces (including dispatch interfaces) and classes can be used in `get_interface_info`.

`test(ComInterface) -> []`

The `test` function simply makes the COM port to a `DebugBreak()` Win32 API call. This breaks into the debugger (such as Visual C++). It is really handy to debug COM interfaces written in C. It is also useful for finding bugs in comet. (Luckily, there are no bugs left in the code.)

`enum(ComInterface) -> ComEnum`

Types:

- ComEnum = ComInterface

This is a utility function that calls the `DISPID_ENUM` property on a COM-object, and returns the result as an interface, suitable for `next` and `nexti`.

`next(ComEnum) -> Variant`
`nexti(ComEnum) -> ComInterface`
`intfenum_next(ComEnum) -> ComInterface`

Types:

- ComEnum = ComInterface

The `next` function calls the `Next` method on an `IEnumVARIANT` interface. The `nexti` function does this too, and also packages the result with `package_interface` (often the `Variant` result is known to be an interface).

The `intfenum_next` calls the `Next` method on an `IEnumIUnknown`, the only difference is the size of the result.

When the iterator reaches the end, an empty tuple {} is returned, this is a value that cannot be in a variant.

`map_enum(ComEnum, Fun)`
`map_enumi(ComEnum, IFun)`
`map_intfenumi(ComEnum, IFun)`

Types:

- ComEnum = ComInterface
- Fun = fun(Variant)
- IFun = fun(ComInterface)

These functions maps over a COM iterator (Com enum) and applies the given fun, the values are collected in a list.

The interface functions (`map_enumi` and `map_intfenumi`) uses `nexti` to iterate. They also releases the interface return from `nexti`. (This means that the value parameter of the fun shouldn't be returned or stored anywhere.)

List of Tables

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.