COM/Automation User Guide and Reference Manual

Version 6.0

Copyright and Trademarks

LispWorks COM/Automation User Guide and Reference Manual

Version 6.0

December 2009

Address

LispWorks Ltd
St. John’s Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England

Telephone

From North America: 877 759 8839 (toll-free)

From elsewhere: +44 1223 421860

Fax

From North America: 617 812 8283

From elsewhere: +44 870 2206189

www.lispworks.com

Contents

Preface

This manual documents the LispWorks COM / Automation API, which provides a toolkit for using Microsoft COM and Automation with Common Lisp.

For details of using OLE and ActiveX controls with the CAPI, see the class **`capi:ole-control-pane`** in the *LispWorks CAPI Reference Manual*.

This preface contains information you need when using the rest of the this manual. It discusses the purpose of this manual, the typographical conventions used, and gives a brief description of the rest of the contents.

Assumptions

The manual assumes that you are familiar with:

- LispWorks
- The LispWorks FLI.
- Common Lisp and CLOS, the Common Lisp Object System
- The functionality of Microsoft COM / Automation.

Unless otherwise stated, examples given in this document assume that the current package has **`COM`** on its package-use-list.

Conventions used in the manual

Throughout this manual, certain typographical conventions have been adopted to aid readability.

Text which refers to Lisp forms is printed `like this`. Variables and values described in the reference sections are printed *like this*.

Entries in the reference sections are listed alphabetically and each entry is headed by the symbol name and type, followed by a number of fields providing further details. These fields consist of a subset of the following: "Summary", "Signature", "Superclasses", "Subclasses", "Slots", "Accessors", "Readers", "Compatibility Note", "Description", "Examples", and "See Also".

Entries with a long "Description" section usually have as their first field a short "Summary" providing a quick overview of the purpose of the symbol being described.

The "Signature" section provides details of the arguments taken by the functions and macros and values returned, separated by the `=>` sign. The top level of parentheses is omitted, but parentheses used for destructuring in macros are included explicitly. Optional items in the syntax of macros are denoted using square brackets *[like this]*. Repeated items have an asterisk suffix like this*.

For classes, only direct sub- and superclasses are detailed in the "Subclasses" and "Superclasses" sections of each entry.

Examples show fragments of code and sometimes the results of evaluating them

Finally, the "See also" section provides a reference to other related symbols.

Please let us know if you find any mistakes in the LispWorks documentation, or if you have any suggestions for improvements.

A Description of the Contents

The manual is divided into three sections, relating to COM, Automation and tools respectively. The COM and Automation sections each contain a user guide and a reference chapter.

Chapter 1, *Using COM* introduces the principles behind the LispWorks COM API and describes how to use it to call COM methods and implement COM servers.

Chapter 2, *COM Reference Entries* provides a detailed description of every function, macro, variable and type in the LispWorks COM API.

Chapter 3, *Using Automation* introduces the LispWorks Automation API and describes how to use it to call Automation methods and implement Automation servers.

Chapter 4, *Automation Reference Entries* provides a detailed description of every function, macro, variable and type in the LispWorks Automation API.

Chapter 5, *Tools* describes some tools which are available in the LispWorks IDE to help with debugging applications using COM / Automation.

1

Using COM

1.1 Prerequisites

Because COM is a low level binary API, many features of the LispWorks COM API depend on the LispWorks FLI. See the *LispWorks Foreign Language Interface User Guide and Reference Manual* for details. You should also have a working knowledge of Microsoft COM.

To compile IDL files, you will need Microsoft® Visual C++® installed.

1.2 Including COM in a Lisp application

1.2.1 Loading the modules

Before using any of the LispWorks COM API, it must be loaded by evaluating

```
(require "com")
```

1.2.2 Generating FLI definitions from COM definitions

COM definitions are typically described in one of two ways, either as IDL files, which allow the full range of COM definitions or as type libraries, which are generally only used for Automation. Before you can use any COM functionality in a Lisp application, you need to convert the COM definitions into

Lisp FLI definitions and various supporting data structures. This corresponds to using **midl.exe** or the MFC Class Wizard when writing C/C++ COM code.

To convert an IDL file, either compile it using the function **midl** or add it to a system definition with the option **:type :midl-file** and compile and load the system.

Note: types like **IDispatch** must declared before they are used, for this conversion to work.

Conversion of type libraries is covered in Chapter 3, "Using Automation".

1.2.3 Standard IDL files

Certain standard IDL files have already been converted to FLI definitions as part of the COM API modules. These are listed below and should not be converted again.

Table 1.1 Pre converted IDL files

IDL file	Part of Lisp module
UNKNWN.IDL	**com**
WTYPES.IDL	**com**
OAIDL.IDL	**automation**
OLEAUTO.IDL	**automation**
OCIDL.IDL	**automation**

1.2.4 Making a COM DLL with LispWorks

You can make a DLL with LispWorks by using **deliver** (or **save-image**) with the **:dll-exports** keyword. The value of the **:dll-exports** keyword can include the keyword **:com**, which exports (with appropriate definitions) the standard four symbols that a COM DLL needs:

```
DllGetClassObject
DllRegisterServer
DllUnregisterServer
DllCanUnloadNow
```

If no other symbols are exported, the value of **:dll-exports** can be the keyword **:com**, which means the same as the list **(:com)**. See the *LispWorks Delivery User Guide* for more details.

1.3 The mapping from COM names to Lisp symbols

COM names are typically a mixture of upper and lower case letters and digits, with words capitalized. These names are mapped to Lisp symbols, adding hyphens to match typical Lisp conventions for word boundaries. These examples illustrate some conversions:

Table 1.2 Examples of COM names and their corresponding Lisp names

COM name	Lisp name
`IUnknown`	`i-unknown`
`pStr`	`p-str`
`DWORD`	`dword`
`IEnumVARIANT`	`i-enum-variant`

In addition, COM methods with the **`propget`** attribute have a **`get-`** prefix added to their names and COM methods with the **`propput`** or **`propputref`** attributes have a **`put-`** prefix added to their names. Note that these prefixes are not used when calling methods via Automation.

To see the mapping for a particular file, look at the output while loading a converted IDL file or type library.

1.4 Obtaining the first COM interface pointer

All interaction with a remote COM server is done via its interface pointers and the most common way to obtain the first interface pointer is using the function **`create-instance`**. This takes the CLSID of the server and returns an interface pointer for the **`i-unknown`** interface unless another interface name is specified.

For example, the following will create an instance of Microsoft Word:

```
(create-instance "000209FF-0000-0000-C000-000000000046")
```

1.5 Reference counting

The lifetime of each COM interface pointer is controlled by its reference count. When a new reference to a COM interface pointer is made, the function **`add-ref`** should be called to increment its reference count. When a reference is removed, the function **`release`** should be called to decrement it again. The macro **with-temp-interface** can be useful when working with temporary interface pointers to ensure that they are released when a body of code exits in any way.

Refer to standard COM texts for more details of the reference counting rules. The LispWorks COM API does not perform any automatic reference counting (sometimes called *smart pointers* in C++).

1.6 Querying for other COM interface pointers

An interface pointer can be queried to discover if the underlying object supports other interfaces. This is done using the function **query-interface**, passing the interface pointer and the **refiid** of the interface to query. A **refiid** is either a foreign pointer to a GUID structure or a **symbol** naming a COM interface as described in Section 1.3.

For example, the function below will find the COM interface pointer for its **i-dispatch** interface:

```
(defun find-dispatch-pointer (ptr)
  (query-interface ptr 'i-dispatch))
```

The macro **with-query-interface** can be used to query an interface pointer and automatically release it again on exit from a body of code.

1.7 Calling COM interface methods

The macros **call-com-interface** and **with-com-interface** are used to call COM methods. To call a COM method, you need to specify the interface name, the method name, a COM interface pointer and suitable arguments. The interface and method names are given as symbols named as in Section 1.3 and the COM interface pointer is a foreign pointer of type **com-interface**. In both macros, the *arg*s and *value*s are as specified in the Section 1.7.1.

The **with-com-interface** macro is useful when several methods are being called with the same COM interface pointer, because it establishes a local macro that takes just the method name and arguments.

For example, the following are equivalent ways of calling the **move** and **resize** methods of a COM interface pointer **window-ptr** for the **i-window** interface:

```
(progn
  (call-com-interface (window-ptr i-window move) 10 10)
  (call-com-interface (window-ptr i-window resize) 100 100))

(with-com-interface (call-window-ptr i-window) window-ptr
  (call-window-ptr move 10 10)
  (call-window-ptr resize 100 100))
```

1.7.1 Data conversion when calling COM methods

All IDL definitions map onto FLI definitions, mirroring the mapping that **midl.exe** does for C/C++. However, IDL provides some additional type information that C/C++ lacks (for instance the **string** attribute), so there are some additional conversions that Lisp performs when it can.

The COM API uses the information from the IDL to convert data between FLI types and Lisp types where appropriate for arguments and return values of COM method calls. In particular:

- Primitive integer types are represented as Lisp integers.
- Primitive char types are represented as Lisp characters.
- Primitive float types are represented as Lisp float types.
- COM interface pointers are FLI objects represented as objects of type **com-interface**, which supports type checking of the interface name.
- Except as detailed below, all other COM types are represented as their equivalent FLI types. This includes other pointer types and structs.

In COM, all parameters have a *direction* which can be either *in*, *out* or both *in* and *out* (referred to as *in-out* here). Arguments and values for client-side COM method calls reflect the direction as described in the following sections. For a complete version of the example code, see the file **examples\com\manual\args\args-calling.lisp** in the LispWorks installation.

1.7.1.1 In parameters

In parameters are passed as positional arguments in the order they are specified and do not affect the return values.

- A parameter with the `string` attribute can be passed either as a foreign pointer or as a Lisp string (converted to a foreign string with dynamic extent for the duration of the call).
- A parameter whose type is either an array type or a pointer type with a `size_is` attribute can be passed either as a foreign pointer or, if the element type is not a foreign aggregate type, as a Lisp array of the appropriate rank (converted to a foreign array with dynamic extent for the duration of the call).
- Otherwise, the Lisp value is converted using the FLI according to the mapping of types defined above.

For example, given the IDL

```
import "unknwn.idl";

[ object,
  uuid(E37A70A0-EFC9-11D5-BF02-000347024BE1)
]
interface IArgumentExamples : IUnknown
{
  typedef [string] char *argString;

  HRESULT inMethod([in] int inInt,
                   [in] argString inString,
                   [in] int inArraySize,
                   [in, size_is(inArraySize)] int *inArray);
}
```

the method `in-method` can be called with Lisp objects like this:

```
(let ((array #(7 6)))
  (call-com-interface (arg-example i-argument-examples
                                   in-method)
                      42
                      "the answer"
                      (length array)
                      array))
```

or with foreign pointers like this:

```
(fli:with-dynamic-foreign-objects ()
  (let* ((farray-size 2)
         (farray (fli:allocate-dynamic-foreign-object
                  :type :int
                  :nelems farray-size
                  :initial-contents '(7 6))))
    (fli:with-foreign-string (fstring elt-count byte-count)
        "the answer"
      (declare (ignore elt-count byte-count))
      (call-com-interface (arg-example i-argument-examples
                                       in-method)
                          42
                          fstring
                          farray-size
                          farray))))
```

Note that the **int** arguments are always passed as Lisp **integer** because **int** is a primitive type.

1.7.1.2 Out parameters

Out parameters are always of type pointer in COM and never appear as positional arguments in the Lisp call. Instead, there is a keyword argument named after the parameter, which can be used to pass an object to be modified by the method. In addition, each *out* parameter generates a return value, which will be **eq** to the value of keyword argument if it was passed and otherwise depends on the type of the parameter as described below.

- If the value of the keyword argument is a foreign pointer then it is passed directly to the method and is expected to point to an object of the appropriate size to contain the returned data.
- If the value of the keyword argument is **nil** then a null pointer is passed to the method.
- Except where specified below, if the keyword argument is omitted, a foreign object with dynamic extent is created to contain the value and a pointer to this object is passed to the method. On return, the contents maybe be converted back to a Lisp object as specified.
- A parameter with the **string** attribute is converted to a Lisp string if the keyword is not passed. If the keyword is passed, the memory for the

string might need to be freed by **co-task-mem-free** if nothing else does this.

- A parameter whose type is either an array type or a pointer type with a **size_is** attribute will be converted to a Lisp array if the keyword is not passed and the element type is not a foreign aggregate type. If the keyword argument is not passed then a new Lisp array is made. If the value of the keyword argument is a Lisp array then that is filled.
- For a parameter whose type is a foreign aggregate type, such as **struct**, the keyword argument must be passed and its value must be as a foreign pointer. This pointer is passed directly to the method.
- For a parameter with the **iid_is** attribute, a **com-interface** pointer is returned using the indicated iid parameter to control the interface name.
- Otherwise, the dynamic extent foreign pointer is dereferenced to obtain the Lisp return value, as if by calling **fli:dereference**.

For example, given the IDL

```
import "unknwn.idl";

[ object,
  uuid(E37A70A0-EFC9-11D5-BF02-000347024BE1)
]
interface IArgumentExamples : IUnknown
{
  typedef [string] char *argString;

  HRESULT outMethod([out] int *outInt,
                    [out] argString *outString,
                    [in] int outArraySize,
                    [out, size_is(outArraySize)] int *outArray);
}
```

the method **out-method** can return Lisp objects like this:

```
(multiple-value-bind (hres int string array)
    (call-com-interface (arg-example i-argument-examples
                                     out-method)
                        8)
  ;; int is of type integer
  ;; string is of type string
  ;; array is of type array
  )
```

or fill an existing array like this:

```
(let ((out-array (make-array 5)))
  (multiple-value-bind (hres int string array)
      (call-com-interface (arg-example i-argument-examples
                                       out-method)
                          (length out-array)
                          :out-array out-array)
    ;; int is of type integer
    ;; string is of type string
    ;; array is eq to out-array and was filled
    ))
```

or set the contents of foreign memory like this:

```
(fli:with-dynamic-foreign-objects ((out-int :int)
                                   (out-string WIN32:LPSTR))
  (let* ((out-farray-size 5)
         (out-farray (fli:allocate-dynamic-foreign-object
                      :type :int
                      :nelems out-farray-size)))
    (multiple-value-bind (hres int string array)
        (call-com-interface (arg-example i-argument-examples
                                         out-method)
                            out-farray-size
                            :out-int out-int
                            :out-string out-string
                            :out-array out-farray)
      ;; Each foreign pointer contains the method's results
      ;; int is the foreign pointer out-int
      ;; string is the foreign pointer out-string
      ;; array is the foreign pointer out-array
      ;; Note that the string must be freed as follows:
      (co-task-mem-free (fli:dereference out-string)))))
```

1.7.1.3 In-out parameters

In-out parameters are always of type pointer in COM and are handled as a mixture of *in* and *out*. In particular, they have both a positional parameter and a keyword parameter, which can be used to control the value passed and conversion of the value returned respectively. Each *in-out* parameter generates a return value, which will be **`eq`** to the value of the keyword argument if it was passed and otherwise depends on the type of the parameter as below.

- As for *out* parameters, if the value of the keyword argument is a foreign pointer then it is passed directly to the method and is expected to be of the appropriate size to contain the returned data. If the value of the keyword argument is **`nil`** then a null pointer is passed to the COM call. The positional argument should be **`nil`** is these cases. If the keyword argument not passed, a foreign object with dynamic extent is created to contain the value, initialized with data from the positional argument before calling the method and possibly converted back to a Lisp value on return.
- For a parameter with the **`string`** attribute, the positional argument is handled as for the *in* argument **`string`** case and the keyword argument is handled as for the *out* argument **`string`** case. The functions **`co-task-mem-alloc`** and **`co-task-mem-free`** should be used to manage the memory for the string itself.
- For a parameter whose type is a non-aggregate array type or a pointer to a non-aggregate type that has the **`size_is`** attribute, the positional argument is handled as for the *in* argument array case and the keyword argument is handled as for the *out* argument array case. To update an existing array, pass it as both the positional and keyword argument values.
- For a parameter whose type is a foreign aggregate type, the keyword argument must be passed and its value must be a foreign pointer. This pointer is passed directly to the method and the positional argument should be **`nil`**.
- Otherwise, a foreign object with dynamic extent is created, set to contain the value of positional argument before calling the method and

dereferenced on return to obtain the Lisp return value, as if by calling **fli:dereference**.

For example, given the IDL

```
import "unknwn.idl";

[ object,
  uuid(E37A70A0-EFC9-11D5-BF02-000347024BE1)
]
interface IArgumentExamples : IUnknown
{
  typedef [string] char *argString;

  HRESULT inoutMethod([in, out] int *inoutInt,
                      [in, out] argString *inoutString,
                      [in] int inoutArraySize,
                      [in, out, size_is(inoutArraySize)]
                      int *inoutArray);
}
```

the method **inout-method** can receive and return Lisp objects like this:

```
(let ((in-array #(7 6)))
  (multiple-value-bind (hres int string array)
      (call-com-interface (arg-example i-argument-examples
                           inout-method)
                          42
                          "the answer"
                          (length in-array)
                          in-array)
    ;; int is of type integer
    ;; string is of type string
    ;; array is of type array
    ))
```

or fill an existing array like this:

```
(let* ((in-array #(7 6))
       (out-array (make-array (length in-array))))
  (multiple-value-bind (hres int string array)
      (call-com-interface (arg-example i-argument-examples
                                       inout-method)
                          42
                          "the answer"
                          (length in-array)
                          in-array
                          :inout-array out-array)
    ;; int is of type integer
    ;; string is of type string
    ;; array is eq to out-array, which was filled
    ))
```

or update an existing array like this:

```
(let* ((inout-array #(7 6)))
  (multiple-value-bind (hres int string array)
      (call-com-interface (arg-example i-argument-examples
                                       inout-method)
                          42
                          "the answer"
                          (length inout-array)
                          inout-array
                          :inout-array inout-array)
    ;; int is of type integer
    ;; string is of type string
    ;; array is eq to inout-array, which was updated
    ))
```

1.7.2 Error handling

Most COM methods return an integer `hresult` to indicate success or failure, which can be checked using `succeeded`,. `s_ok`, `hresult-equal` or `check-hresult`.

In addition, after calling a COM method that provides extended error information, you can call the function `get-error-info` to obtain more details of any error that occurred. This is supplied with a list of *fields*, which should be keywords specifying the parts of the error information to obtain.

For example, in the session below, `tt` is a COM interface pointer for the `i-test-suite-1` interface:

```
CL-USER 186 > (call-com-interface (tt i-test-suite-1 fx))

"in fx"          ;; implementation running
-2147352567      ;; the error code DISP_E_EXCEPTION

CL-USER 187 > (get-error-info :fields '(:description
                                        :source))
("foo" "fx")

CL-USER 188 >
```

1.8 Implementing COM interfaces in Lisp

Lisp implementations of COM interfaces are created by defining an appropriate class and then defining COM methods for all the interfaces implemented by this class.

The class can inherit from **`standard-i-unknown`** to obtain an implementation of the **`i-unknown`** interface. This superclass provides reference counting and an implementation of the **`query-interface`** method that generates COM interface pointers for the interfaces specified in the class definition. It also supports *aggregation*.

There are two important things to note about COM classes and methods:

- The implementation objects and COM interface pointers are different things: an interface pointer must be queried from the implementation object explicitly and the function **`com-object-from-pointer`** can be used to obtain an object from an interface pointer. This is show in Figure 1.1 below.
- COM methods are not defined with **`defmethod`** because they have very specific conventions for passing arguments and returning values that are different from those of Lisp.

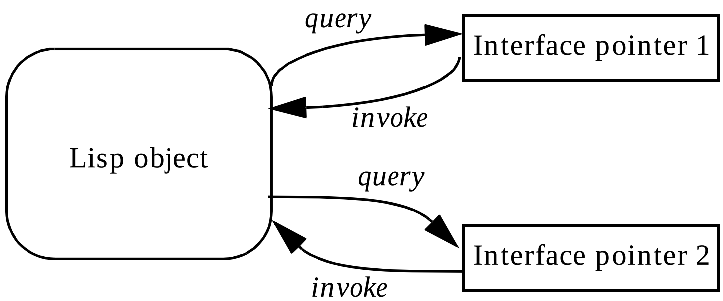


Figure 1.1 The relationship between an Lisp object and its COM interface pointers

1.8.1 Steps required to implement COM interfaces

To implement a COM interface in Lisp, you need the following:

1. Some COM interface definitions, converted to Lisp as specified in Section 1.2.2
2. A COM object class defined with the macro **define-com-implementation**, specifying the interface(s) to implement.
3. Implementations of the methods using **define-com-method**.
4. If the objects are to be created by another process, a description of the class factories created with **make-factory-entry** and registered with **register-class-factory-entry**.
5. Initialization code to call **co-initialize**. It should also call **start-factories** in a thread that will be processing Windows messages (for instance a CAPI thread) if you have registered class factories.

1.8.2 The lifecycle of a COM object

Since COM objects can be accessed from outside the Lisp world, possibly from a different application, their lifetimes are controlled more carefully than those of normal Lisp objects. The diagram below shows the lifecycle of a typical COM object.

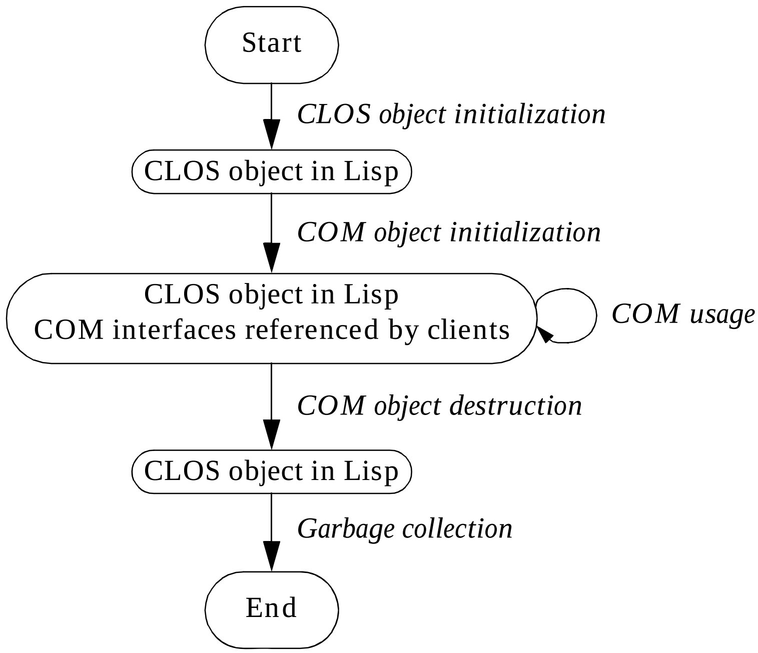


Figure 1.2 The lifecyle of a COM object

Each COM object goes through the following stages.

1. **CLOS object initialization.**

   In the first stage, the object is created by a call to `make-instance`, either by a class factory (see Section 1.8.3) or explicitly by the application. The normal CLOS initialization mechanisms such as `initialize-instance` can be used to initialize the object. During this stage, the object is known only to Lisp and can be garbage collected if the next stage is not reached.

2. **COM initialization.**

   At some point, the server makes the first COM interface pointer for the object by invoking the COM method `query-interface`, either automatically in the class factory or explicitly using by using macros such as `query-object-interface` or `call-com-object`. When this happens, the object's reference count will become 1 and the object will be stored in the

COM runtime system. In addition, the generic function **com-object-initialize** is called to allow class-specific COM initialization to be done.

3. **COM usage.**

   In this stage, the object is used via its COM interface pointers by a client or directly by Lisp code in the server. Several COM interface pointers might be created and each one contributes to the overall reference count of the object.

4. **COM destruction.**

   This stage is entered when the reference count is decremented to zero, which is triggered by all the COM interface pointers being released by their clients. The generic function **com-object-destructor** is called to allow class-specific COM cleanups and the object is removed from the COM runtime system. From now on, the object is not known to COM world.

5. **Garbage collection.**

   The final stage of an object's lifecyle is the normal Lisp garbage collection process, which removes the object from memory when there are no more references to it.

1.8.3 Class factories

The LispWorks COM runtime provides an implementation of the *class factory* protocol, which will construct COM objects on demand. The class factory implementation supports *aggregation* when passed an outer unknown pointer.

Class factories are described by objects created with **make-factory-entry** and must be registered with the COM runtime using **register-class-factory-entry**. The function **start-factories** should be called when the application initializes to start all the registered class factories.

When using the Automation API described in Chapter 3 and Chapter 4, class factories are created and registered automatically by the **define-automation-component** macro if appropriate.

1.8.4 Unimplemented methods

If the class does not define all the COM methods for the interfaces it implements, then some of those methods may be inherited from superclasses (see Section 1.8.5). If there is no direct or inherited definition of a method, then a default method that returns **E_NOTIMPL** will be provided automatically. The default method also fills all *out* arguments with null bytes and ignores all *in* and *in-out* arguments except those needed to compute the size of arrays for filling *out* arguments.

1.8.5 Inheritance

A COM object class will inherit COM method implementations from its superclasses if no direct method is defined. However, unlike Lisp methods where an effective method is computed from the set of applicable methods for each generic function, COM methods are always inherited in groups via their defining interface. This is because the interface is used to call a COM method, not the COM object

Specifically, each method is inherited from the first class in the class precedence list that implements the interface where the method is declared. No attempt is made to search further down the class precedence list if this class is using the unimplemented method definition described in Section 1.8.4.

1.8.5.1 An example of multiple inheritance

The inheritance rules may lead to unexpected results in the case of multiple inheritance. For example, consider the following IDL:

```
// IDL definition of IFoo
import "unknwn.idl";

[ uuid(7D9EB760-E4E5-11D5-BF02-000347024BE1) ]
interface IFoo : IUnknown
{
  HRESULT meth1();
  HRESULT meth2();
  HRESULT meth3();
}
```

and these three (partial) implementations of the interface **i-foo**.

1. An implementation with no definition of **meth2**:

```
(define-com-implementation foo-impl-1 ()
  ()
  (:interfaces i-foo))

(define-com-method meth1 ((this foo-impl-1))
  s_ok)

(define-com-method meth3 ((this foo-impl-1))
  s_ok)
```

2. An implementation with no definition except **meth2**:

```
(define-com-implementation foo-impl-2 ()
  ()
  (:interfaces i-foo))

(define-com-method meth2 ((this foo-impl-2))
  s_ok)
```

3. A combined implementation, inheriting from steps **1** and **2**.

```
(define-com-implementation foo-impl-12 (foo-impl-1
                                         foo-impl-2)
  ()
  (:interfaces i-foo))
```

In step **3**, the class **foo-impl-12** implements the interface **i-foo**, but inherits all the **i-foo** method definitions from **foo-impl-1**, which is the first class in the class precedence list that implements that interface. These method definitions include the "unimplemented" definition of **meth2** in **foo-impl-1**, which hides the definition in the other superclass **foo-impl-2**. As a result, when the following form is evaluated with **p-foo** created from an instance of **foo-impl-12**:

```
(let ((object (make-instance 'foo-impl-12)))
  (with-temp-interface (p-foo)
      (nth-value 1 (query-object-interface
                    foo-impl-12
                    object
                    'i-foo))
    (with-com-interface (call-p-foo i-foo) p-foo
      (values (call-p-foo meth1)
              (call-p-foo meth2)
              (call-p-foo meth3)))))
```

the three values are `S_OK`, `E_NOTIMPL` and `S_OK`.

1.8.5.2 A second example of multiple inheritance

Here is a further extension to the example in Section 1.8.5.1, with an additional interface `i-foo-ex`.that inherits from `i-foo` as in the following IDL:

```
[ uuid(7D9EB761-E4E5-11D5-BF02-000347024BE1) ]
interface IFooEx : IFoo
{
  HRESULT meth4();
}
```

This interface has the following additional implementations:

1. An implementation defining all the methods in `i-foo-ex`:

```
(define-com-implementation foo-ex-impl-1 ()
  ()
  (:interfaces i-foo-ex))

(define-com-method meth1 ((this foo-ex-impl-1))
  s_ok)

(define-com-method meth2 ((this foo-ex-impl-1))
  s_ok)

(define-com-method meth3 ((this foo-ex-impl-1))
  s_ok)

(define-com-method meth4 ((this foo-ex-impl-1))
  s_ok)
```

2. A combined implementation, inheriting from step **3** from Section 1.8.5.1 and step **1** above.

```
(define-com-implementation foo-ex-impl-2 (foo-impl-12
                                           foo-ex-impl-1)
  ()
  (:interfaces i-foo-ex))
```

In step **2**, the class `foo-ex-impl-2` implements the interface `i-foo-ex` and is a subclass of `foo-ex-impl-1`, which implements `i-foo`. When the following form is evaluated with `p-foo-ex` created from an instance of `foo-ex-impl-2`:

```
(let ((object (make-instance 'foo-ex-impl-2)))
  (with-temp-interface (p-foo-ex)
      (nth-value 1 (query-object-interface
                    foo-ex-impl-2
                    object
                    'i-foo-ex))
    (with-com-interface (call-p-foo i-foo-ex) p-foo-ex
      (values (call-p-foo meth1)
              (call-p-foo meth2)
              (call-p-foo meth3)
              (call-p-foo meth4)))))
```

the four values are **S_OK**, **E_NOTIMPL**, **S_OK** and **S_OK**.

Note that, even though **foo-ex-impl-2** only explicitly implements **i-foo-ex**, the methods **meth1**, **meth2** and **meth3** were declared in its parent interface **i-foo**. This means that their definitions (including the "unimplemented" definition of **meth2**) are inherited from **foo-impl** (via **foo-impl-12**), because **foo-impl-12** is before **foo-ex-impl-2** in the class precedence list of **foo-ex-impl-2**. Only **meth4**, which is declared in **i-foo-ex**, is inherited from **foo-ex-impl-1**.

1.8.6 Data conversion in define-com-method

All IDL definitions map onto FLI definitions, mirroring the mapping that **midl.exe** does for C/C++. However, IDL provides some additional type information that C/C++ lacks (for instance the **string** attribute), so there are some additional conversions that Lisp performs when it can. For a complete example of data conversion, see the file **examples\com\manual\args\args-impl.lisp** in the LispWorks installation.

1.8.6.1 FLI types

The COM API uses the information from the IDL to convert data between FLI types and Lisp types where appropriate for arguments and return values of COM method definitions. In particular:

- Primitive integer types are represented as Lisp integers
- Primitive char types are represented as Lisp characters.
- Primitive float types are represented as Lisp float types.

- COM interface pointers are represented as objects of type **com-interface**, which supports type checking of the interface name.
- All other types are represented as their equivalent FLI types. This includes other pointer types and structs.

Each argument is the IDL has a corresponding argument in the **define-com-method** form. In addition, each argument has a *pass-style* which specifies whether additional conversions are performed.

If the *pass-style* of a parameter is **:foreign**, then the value will be exactly what the FLI would provide, i.e. foreign pointers for strings and for all *out* or *in-out* parameters (which are always pointers in the IDL).

If the *pass-style* of a parameter is **:lisp**, then the conversions described in the following sections will be done.

1.8.6.2 In parameters

For *in* parameters:

- A parameter with the **string** attribute will be converted to a Lisp string. The string should not be destructively modified by the body.
- A parameter of COM type **BSTR** will be converted to a Lisp string. The string should not be destructively modified by the body.
- A parameter of COM type **VARIANT*** will be converted to a Lisp object according to the VT code in the variant (see Table 3.1, page 81).
- A parameter of COM type **SAFEARRAY(***type***)** or **SAFEARRAY(***type***)*** will be converted to a Lisp array. The elements of type *type* are converted as in Table 3.1.
- A parameter of COM type **VARIANT_BOOL** will be converted to **nil** (for zero) or **t** (for any other value). Note that a parameter of type **BOOL** will be converted to an **integer** because type libraries provide no way to distinguish this case from the primitive integer type.
- A parameter whose type is an array type or a pointer type with a **size_is** attribute will be converted to a temporary Lisp array. The Lisp array might have dynamic extent.

- Otherwise, the value is converted to a Lisp value using the FLI according to the mapping of types defined in Section 1.8.6.1.

1.8.6.3 Out parameters

For *out* parameters:

- A parameter whose type is an array type or a pointer type with a **size_is** attribute will be converted to a Lisp array of the appropriate size allocated for the dynamic extent of the body forms. After the body has been evaluated, the contents of the array will be copied into the foreign array that the caller has supplied.
- For other types, the parameter will be **nil** initially and the body should use **setq** to set it to the value to be returned.

In the latter case, the value will be converted to a foreign object after the body has been evaluated. The following conversions are done:

- For a parameter with the **string** attribute, a Lisp string will be converted to a foreign string using **CoTaskMemAlloc()**.
- For a parameter of COM type **BSTR***, a Lisp string will be converted to a foreign string using **SysAllocString()**.
- For a parameter of COM type **VARIANT***, the value can be any Lisp value, with the VT code being set according to the Lisp type (see Table 3.1, page 81). If exact control is required, use the *pass-style* **:foreign** and the function **set-variant**.
- For a parameter of COM type **SAFEARRAY(***type***)***, the value can be either a foreign pointer to an appropriate **SAFEARRAY** or a Lisp array. In the latter case, a new **SAFEARRAY** is created which contains the elements of the Lisp array converted as in Table 3.1.
- For a parameter of COM type **VARIANT_BOOL***, the value can be a generalized boolean.
- Otherwise, the Lisp value will be converted using the FLI according to the mapping of types defined in Section 1.8.6.1.

1.8.6.4 In-out parameters

For *in-out* parameters:

- A parameter whose type is an array type or a pointer type with a **size_is** attribute will be converted to a Lisp array of the appropriate size allocated for the dynamic extent of the body forms. The initial contents of the Lisp array will be taken from the foreign array which was passed by the caller. After the body has been evaluated, the contents of the Lisp array will be copied back into the foreign array.
- For a parameter with the **string** attribute, the parameter will be the converted to a Lisp string. To return a different string, the parameter should be set to another (non **eq**) Lisp string, which will cause the original foreign string to be freed with **CoTaskMemFree()** and a new foreign string allocated with **CoTaskMemAlloc()**. The initial string should not be destructively modified by the body.
- For a parameter of COM type **BSTR***, the parameter will be the converted to a Lisp string. To return a different string, the parameter should be set to another (non **eq**) Lisp string, which will cause the original foreign string to be freed with **SysFreeString()** and a new foreign string allocated with **SysAllocString()**.
- For parameters of COM type **VARIANT***, the parameter will be converted to a Lisp object (see Table 3.1, page 81). To return a different value, the parameter should be set to another (non **eq**) value, which will be placed back into the **VARIANT** with the VT code being set according to the Lisp type (see Table 3.1, page 81). If exact control of the VT code is required, use the *pass-style* **:foreign** and the function **set-variant**.
- For parameters of COM type **SAFEARRAY(***type***)***, the parameter will be converted to a Lisp array. The elements of type *type* are converted as in Table 3.1. To return a different value, the parameter should be set to another (non **eq**) value, which can be either a foreign pointer to an appropriate **SAFEARRAY** or a Lisp array. In the latter case, a new **SAFEARRAY** is created which contains the elements of the Lisp array converted as in Table 3.1.
- For parameter of COM type **VARIANT_BOOL***, the parameter will be **nil** or **t** according to the initial value (zero or non zero). To return a differ-

ent value, set the parameter to a new value, which can be a generalized boolean.

1.9 Calling COM object methods from Lisp

Within the implementation of a COM object, the macros **`call-com-object`** and **`with-com-object`** can be used to call COM methods directly for a COM object without using an interface pointer. To call a COM method, you need to specify the class name, the method name, the interface name if the method name is not unique, a COM object and suitable arguments. The class name is a symbol as used in the **`define-com-implementation`** form and can be a superclass of the actual object class. The method and interface names are given as symbols named as in Section 1.3. and the arguments and values are as specified below in Section 1.9.1. These macros should be used with caution because they assume that the caller knows the implementation's *pass-style* for all the arguments.

The **`with-com-object`** macro is useful when several methods are being called with the same COM object, because it establishes a local macro that takes just the method name and arguments.

1.9.1 Data conversion when calling COM object methods

No explicit argument or return value conversion is done by **`call-com-object`** or **`with-com-object`**. As a result, every argument must be passed as a positional argument and must be of the type expected by the method's implementation The allowable types are described in the following sections.

1.9.1.1 In parameters

For *in* parameters,

- For a parameter with the **`string`** attribute, the value can be a Lisp string.
- For a parameter of COM type **`BSTR`**, the value can be a Lisp string.
- For a parameter whose type is an array type or a pointer type with a **`size_is`** attribute, the value can be a Lisp array of the appropriate rank and dimension.

- Otherwise, the value should match what the FLI would generate for the parameter's type.

1.9.1.2 Out parameters

For *out* parameters,

- If **nil** is passed, the value from the method is returned without any conversion.
- For a parameter whose type is an array type or a pointer type with a **size_is** attribute, the value can be a Lisp array. The contents of the array will be modified by the method and the array will be returned as a value.
- Otherwise, the value should be a foreign pointer of the type that the FLI would generate for the parameter's type. The foreign pointer will be returned as a value.

1.9.1.3 In-out parameters

For *in-out* parameters,

- For a parameter whose type is an array type or a pointer type with a **size_is** attribute, the value can be a Lisp array. The contents of the array will be modified by the method and the array will be returned as a value.
- For a parameter with the **string** attribute, the parameter can be a Lisp string. The value of the parameter at the end of the body will be returned as a value.
- For a parameter of COM type **BSTR***, the parameter can be a Lisp string. The value of the parameter at the end of the body will be returned as a value.
- For parameters of COM type **VARIANT***, the parameter can be any Lisp object. The value of the parameter at the end of the body will be returned as a value.
- If the value is a foreign pointer of the type that the FLI would generate for the parameter's type then the foreign object it points to will be the

value of the parameter. The foreign pointer will be returned as a value, with the new contents as modified (or not) by the method.

- Otherwise, the parameter is passed directly to the method and the value of the parameter at the end of the body will be returned as a value.

2

COM Reference Entries

The following chapter documents COM functionality.

add-ref *Function*

Summary Increments the reference count of a COM interface pointer.

Package `com`

Signature `add-ref` ***interface-ptr*** `=>` ***ref-count***

Arguments *interface-ptr* A COM interface pointer.

Values *ref-count* The new reference count.

Description Each COM interface pointer has a reference count which is used by the server to control its lifetime. The function `add-ref` should be called whenever an extra reference to the interface pointer is being made. The function invokes the COM method `IUnknown::AddRef` so the form `(add-ref ptr)` is equivalent to using `call-com-interface` as follows:

```
(call-com-interface (ptr i-unknown add-ref))
```

Example

```
(add-ref p-foo)
```

See also

release
interface-ref
query-interface
call-com-interface

call-com-interface *Macro*

Summary Invokes a method from a particular COM interface.

Package **com**

Signature **call-com-interface** ***spec arg**** **=>** ***values***

spec **::=** **(*interface-ptr interface-name method-name*)**

Arguments

spec	The interface pointer and a specification of the method to be called.
interface-ptr	A form which is evaluated to yield a COM interface pointer.
interface-name	A symbol which names the com interface. It is not evaluated.
method-name	A symbol which names the method. It is not evaluated.
arg	Arguments to the method (see Section 1.7.1, “Data conversion when calling COM methods” for details).

Values

values	Values from the method (see Section 1.7.1, “Data conversion when calling COM methods” for details).

Description The macro **`call-com-interface`** invokes the method *method-name* for the COM interface *interface-name*, which should the type or a supertype of the actual type of *interface-ptr*. The *args* and *values* are described in detail in Section 1.7.1, “Data conversion when calling COM methods”.

Example This example invokes the COM method **`GetTypeInfo`** in the interface **`IDispatch`**.

```
(defun get-type-info (disp tinfo &key
                      (locale LOCALE_SYSTEM_DEFAULT))
  (multiple-value-bind (hres typeinfo)
      (call-com-interface
          (disp i-dispatch get-type-info)
          tinfo locale)
    (check-hresult hres 'get-type-info)
    typeinfo))
```

See also **`with-com-interface`**
`query-interface`
`add-ref`
`release`

call-com-object *Macro*

Summary Invokes a COM method on a COM object.

Package **`com`**

Signature **`call-com-object`** ***spec arg**** **`=>`** ***values***

spec **`::=`** **`(`*****object class-name method-spec*** **`&key`** ***interface*****`)`**

method-spec **`::=`** ***method-name*** | **`(`*****interface-name method-name*****`)`**

Arguments *spec* The object and a specification of the method to be called.

object A form which is evaluated to yield a COM object.

class-name	A symbol which names the COM implementation class. It is not evaluated.
method-spec	Specifies the method to be called. It is not evaluated.
method-name	A symbol naming the method to call.
interface-name	A symbol naming the interface of the method to call. This is only required if the implementation class *class-name* has more than one method with the given *method-name*.
interface	An optional form which when evaluated should yield a COM interface pointer. This is only needed if the definition of the method being called has the *interface* keyword in its *class-spec*.
arg	Arguments to the method (see Section 1.9.1, “Data conversion when calling COM object methods” for details).

Values

values	Values from the method (see Section 1.9.1, “Data conversion when calling COM object methods” for details).

Description

The macro **`call-com-object`** invokes the method *method-name* for the COM class *class-name*, which should the type or a supertype of the actual type of *object*. The *arg*s and *values* are described in detail in Section 1.9.1, “Data conversion when calling COM object methods”.

Note that, because this macro requires a COM object, it can only be used by the implementation of that object. All other code should use **`call-com-interface`** with the appropriate COM interface pointer.

Examples

```
(call-com-object (my-doc doc-impl move) 0 0)
```

```
(call-com-object (my-doc doc-impl resize) 100 200)
```

See also
with-com-object
query-object-interface
call-com-interface

check-hresult *Macro*

Summary
Signals an error if a result code indicates a failure.

Package
com

Signature
check-hresult ***hresult function-name***

Arguments

hresult An integer **hresult**.

function-name A name for inclusion in the error message.

Description
The **check-hresult** macro checks the *hresult* and returns if the it is one of the 'succeeded' values, for instance **S_OK** or **S_FALSE**. Otherwise it signals an error of type **com-error**, which will include the *function-name* in its message.

Examples

```
(check-hresult S_OK "test") => nil
```

```
(check-hresult E_NOINTERFACE "test")
signals an error mentioning "test"
```

See also
succeeded
hresult
hresult-equal

co-initialize *Function*

Summary
Initialize the COM library in the current thread.

Package
com

Signature `co-initialize &optional` ***co-init***

Arguments *co-init* Flags to specify the concurrency model and initialization options for the thread.

Description The function **`co-initialize`** initializes COM for the current thread. This must be called by every thread that uses COM client or server functions.

The default value of *co-init* is **`COINIT_APARTMENTTHREADED`**. Other flags are allowed as for the *dwCoInit* argument to **`CoInitializeEx`**.

Examples

```
(co-initialize)
```

See also **`co-uninitialize`**

co-task-mem-alloc *Function*

Summary Allocates a block of foreign memory for use in COM method argument passing.

Package **`com`**

Signature `co-task-mem-alloc &key` ***type pointer-type initial-element initial-contents nelems*** `=>` ***pointer***

Arguments

type A FLI type specifying the type of the object to be allocated. If *type* is supplied, *pointer-type* must not be supplied.

pointer-type A foreign pointer type specifying the type of the pointer object to be allocated. If *pointer-type* is supplied, *type* must not be supplied.

initial-element A keyword setting the initial value of every element in the newly allocated object to *initial-element*.

initial-contents	A list of forms which initialize the contents of each element in the newly allocated object.
nelems	An integer specifying how many copies of the object should be allocated. The default value is 1.

Values *pointer* A pointer to the specified *type* or *pointer-type*.

Description The function **co-task-mem-alloc** calls the C function **CoTaskMemAlloc()** to allocate a block of memory. The various arguments are handled in the same way as for the function **fli:allocate-foreign-object** (see the *LispWorks Foreign Language Interface User Guide and Reference Manual*).

Examples Two ways to allocate memory for an integer:

```
(co-task-mem-alloc :type :int)
```

```
(co-task-mem-alloc :pointer-type '(:pointer :int))
```

See also **co-task-mem-free**

co-task-mem-free *Function*

Summary Frees a block of foreign memory used in COM method argument passing.

Package **com**

Signature **co-task-mem-free** ***pointer*** **=>** ***pointer2***

Arguments *pointer* A foreign pointer for the block to be freed.

Values *pointer2* The same as *pointer*.

Description The function **`co-task-mem-free`** calls the C function **`CoTaskMemFree()`** to free a block of memory. The pointer should not be dereferenced after calling this function.

Example

```
(co-task-mem-free ptr)
```

See also **`co-task-mem-alloc`**

co-uninitialize *Function*

Summary Close the COM library in the current thread.

Package **`com`**

Signature **`co-uninitialize`**

Description The function **`co-uninitialize`** closes the COM library on the current thread. This should be called when COM is no longer required, for instance before exiting the application.

Examples

```
(co-uninitialize)
```

See also **`co-initialize`**

com-error *Condition Class*

Summary The condition class used to signal errors from COM.

Package **`com`**

Superclasses **`error`**

Initargs

`:hresult` An integer giving the **`hresult`** of the error.

`:function-name`

Either **nil** or a string or symbol describing the function that generated the error.

Readers **com-error-hresult**
com-error-function-name

Description The class **com-error** is used by the Lisp COM API when signalling errors that originate as **hresult** code from COM.

Example This function silently ignores the **E_NOINTERFACE** error:

```
(defun call-ignoring-nointerface-error (function)
  (handler-bind
      ((com-error
        #'(lambda (condition)
            (when (hresult-equal (com-error-hresult
                                   condition)
                                 E_NOINTERFACE)
              (return-from
                  call-ignoring-nointerface-error
                nil)))))
    (funcall function)))
```

See also **check-hresult**
hresult-equal
hresult

com-interface *Class*

Summary The class of all COM interface pointers.

Package **com**

Superclasses **fli:foreign-pointer**

Description The class **com-interface** is used for all COM interface pointers.

Example
```
(typep (query-interface ptr 'i-unknown) 'com-interface)
=> t
```

See also `call-com-interface`

com-interface-refguid *Function*

Summary Return the `refguid` object for a named COM interface.

Package `com`

Signature `com-interface-refguid` ***interface-name*** `=>` ***refguid***

Arguments *interface-name* A symbol naming a COM interface.

Values *refguid* The `refguid` object matching *interface-name*.

Description The function `com-interface-refguid` returns a `refguid` object that matches *interface-name*, which should be a symbol as described in Section 1.3, “The mapping from COM names to Lisp symbols”. This definition of this COM interface must have been converted to Lisp FLI definitions as in Section 1.2.2, “Generating FLI definitions from COM definitions” or Section 3.1, “Including Automation in a Lisp application”.

Examples
```
(guid-to-string (com-interface-refguid 'i-unknown))
=> "00000000-0000-0000-C000-000000000046"
```

See also
`refguid`
`guid-equal`
`guid-to-string`
`make-guid-from-string`
`refguid-interface-name`

com-object *Class*

Summary The ancestor of an COM object implementation classes.

Package `com`

Superclasses `standard-object`

Subclasses `standard-i-unknown`

Description The class `com-object` is the ancestor of all COM object implementation classes. In general, it is more useful to inherit from its subclass `standard-i-unknown`, which provides an implementation of the `i-unknown` interface.

Example For a COM object `my-doc`:

```
(typep my-doc 'com-object) => t
```

See also `standard-i-unknown`

com-object-destructor *Generic Function*

Summary Called when a COM object loses its last interface pointer.

Package `com`

Signature `com-object-destructor` ***object***

Arguments *object* A COM object.

Method Signatures

```
com-object-destructor (object standard-i-unknown)

com-object-destructor :around
                      (object standard-i-unknown)
```

Description The generic function `com-object-destructor` is called by the implementation of the class `standard-i-unknown` at the point where the last COM interface pointer is removed for the object, i.e. where the overall reference count becomes zero. After this, the object is known only to Lisp and is not involved in any COM operations and will be freed as normal by the garbage collector. The built-in primary method specializing on **standard-i-unknown** does nothing. The build-in around method specializing on **standard-i-unknown** frees the memory used by the COM interface pointers. Typically, after methods are defined to handle class-specific cleanups.

This function should not be called directly by user code.

Examples

```
(defmethod com-object-destructor :after
                                  ((my-doc doc-impl))
  (close (document-file my-doc)))
```

See also

com-object-initialize
standard-i-unknown

com-object-from-pointer *Function*

Summary
Return the COM object that implements a particular COM interface pointer.

Package
com

Signature
com-object-from-pointer ***pointer*** **=>** ***object***

Arguments

pointer	A foreign pointer.

Values

object	*A COM object or nil.*

Description
The function **com-object-from-pointer** returns the COM object that implements pointer. The value of *pointer* should be a foreign pointer or COM interface pointer that was created by LispWorks itself and implemented by a subclass of **com-**

object. If *pointer* is not a known COM interface pointer then **nil** is returned.

Example `(com-object-from-pointer my-ptr)`

See also **com-object**

com-object-initialize *Generic Function*

Summary Called when a COM object gets its first interface pointer.

Package **com**

Signature **com-object-initialize** ***object***

Arguments *object* A COM object.

Method Signatures **com-object-initialize (*object* standard-i-unknown)**

Description The generic function **com-object-initialize** is called by the built-in class **standard-i-unknown** at the point where the first COM interface pointer is made for the object. Prior to this, the object is known only to Lisp and is not involved in any COM operations. The built-in primary method specializing on **standard-i-unknown** does nothing.

This function should not be called directly by user code.

Examples

```
(defmethod com-object-initialize :after
                                  ((my-doc doc-impl))
  (ensure-open-document-file my-doc))
```

See also **com-object-destructor**
standard-i-unknown

com-object-query-interface *Generic Function*

Summary Called by the built in implementation of **`query-interface`**.

Package **`com`**

Signature **`com-object-query-interface`** ***object iid***

Arguments

object A COM object.

iid A GUID foreign pointer.

Method Signatures **`com-object-query-interface (`*object* `standard-i-unknown)`**
`(`*iid* `t)`

Description The generic function **`com-object-query-interface`** is called by the built-in implementation of **`query-interface`** for the class **`standard-i-unknown`**. The built-in primary method specializing on **`standard-i-unknown`** handles the **`i-unknown`** interface and all the interfaces specified by the **`define-com-implementation`** form for the class of *object*.

In most cases, there is no need to specialize this generic function for user-defined classes.

This function should not be called directly by user code.

See also **`define-com-implementation`**
`standard-i-unknown`

create-instance *Function*

Summary Starts the implementation of a remote COM object and returns its interface pointer.

Package **`com`**

Signature **`create-instance`** ***clsid*** **`&key`** ***unknown-outer clsctx riid errorp***

=> *interface-ptr*

Arguments

	clsid	A string or a **refguid** giving a CLSID to create.
	unknown-outer	A COM interface pointer specifying the outer **i-unknown** if the new instance is to be *aggregated*.
	clsctx	A CLSCTX value, which defaults to **CLSCTX_SERVER**.
	riid	An optional **refiid** giving the COM interface name to return. If **nil**, then **i-unknown** is used.
	errorp	A boolean. The default is **t**.

Values

	interface-ptr	A COM interface pointer for *riid*.

Description

Creates an instance of the COM server associated with *clsid* and returns an interface pointer for its *riid* interface.

If the server cannot be started, then an error of type **com-error** will be signalled if *errorp* is true, otherwise **nil** will be returned.

If *unknown-outer* is non-nil, it will be passed as the outer unknown interface to be aggregated with the new instance.

Note: to create an **i-dispatch** interface and set an event handler, you can use **create-instance-with-events**.

Example

```
(create-instance
 "000209FF-0000-0000-C000-000000000046")
```

See also

refguid
refiid
i-unknown
create-object
create-instance-with-events

define-com-implementation *Macro*

Summary Defines an implementation class for a particular set of interfaces.

Package `com`

Signature **`define-com-implementation`** ***class-name*** **(*superclass-name**)** **(*slot-specifier**)** ***class-option****

Arguments

class-name	A symbol naming the class to define.
superclass-name	A symbol naming a superclass to inherit from.
slot-specifier	A slot description as used by **`defclass`**.
class-option	An option as used by **`defclass`**.

Description The macro **`define-com-implementation`** defines a **`standard-class`** which is used to implement a COM object. Normal **`defclass`** inheritance rules apply for slots and Lisp methods.

Each *superclass-name* argument specifies a direct superclass of the new class, which can be another COM implementation class or any other **`standard-class`** provided that **`com-object`** is included somewhere in the overall class precedence list. To get the built-in handling for the **`i-unknown`** interface, inherit from **`standard-i-unknown`** (which is the default superclass if no others are specified).

The *slot-specifier*s are standard **`defclass`** slot definitions.

The *class-option*s are standard defclass options. In addition the following options are recognized:

`(:interfaces` ***interface-name****`)`**

Each *interface-name* specifies a COM interface that the object will implement. **`i-unknown`** should not be specified unless the you wish to replace the standard implementation provided by **`standard-i-unknown`**. If more than one *interface-name* is given then all the methods must have different names (except for those which are inherited from a common parent interface).

`(:inherit-from` ***class-name interface-name*`*)`**

This indicates that the class will inherit the implementation of all the methods in the interfaces specified by the *interface-names* directly from *class-name*. The *class-name* must be one of the direct or indirect superclasses of the class being defined. Without this option, methods from superclasses are inherited indirectly and can be shadowed in the class being defined. Use of **`:inherit-from`** allows various internal space-optimizations.

For example, given a COM class **`foo-impl`** which implements the **`i-foo`** interface, this definition of **`bar-impl`**:

```
(define-com-implementation bar-impl (foo-impl)
    ()
    (:interfaces i-foo))
```

will allow methods from **`i-foo`** to be shadowed whereas this definition:

```
(define-com-implementation bar-impl (foo-impl)
    (:interfaces i-foo)
    (:inherit-from foo-impl i-foo))
```

will result in an error if a method from **`i-foo`** is redefined for **`bar-impl`**.

`(:dont-implement` ***interface-name*`*)`**

This option tells **`standard-i-unknown`** that it should not respond to **`query-interface`** for the given *interface-name*s (which should be parents of the interfaces implemented by the class being defined). Normally, **`standard-i-unknown`** will respond to **`query-interface`** for a parent interface by returning a pointer to the child interface.

For example, given an interface **`i-foo-internal`** and sub-interface **`i-foo-public`**, the following definition

```
(define-com-implementation foo-impl ()
    ()
    (:interfaces i-foo-public))
```

specifies that **`foo-impl`** will respond to **`query-interface`** for **`i-foo-public`** and **`i-foo-internal`**, whereas the following definition

```
(define-com-implementation foo-impl ()
    (:interfaces i-foo-public)
    (:dont-implement i-foo-internal))
```

specifies that **`foo-impl`** will respond to **`query-interface`** for **`i-foo-public`** only.

Examples

```
(define-com-implementation i-robot-impl ()
  ((tools :accessor robot-tools))
  (:interfaces i-robot)
  )

(define-com-implementation i-r2d2-impl (i-robot-impl)
  ()
  (:interfaces i-robot i-r2d2)
  )
```

See also

`define-com-method`
`standard-i-unknown`

define-com-method *Macro*

Summary

The **`define-com-method`** macro is used to define a COM method for a particular implementation class.

Package `com`

Signature

`define-com-method` ***method-spec*** `(`***class-spec arg-spec****`)`
form*

method-spec `::=` ***method-name*** | `(`***interface-name method-name***`)`

class-spec `::=` `(`***this class-name*** `&key` ***interface***`)`

arg-spec `::=` `(`***parameter-name [direction [pass-style]]***`)`

Arguments

method-spec	Specifies the method to be defined.
method-name	A symbol naming the method to define.
interface-name	A symbol naming the interface of the method to define. This is only required if the implementation class *class-name* has more than one method with the given *method-name*.
class-spec	Specifies the implementation class and variables bound to the object with in the *forms*.
this	A symbol which will be bound to the COM object whose method is being invoked.
class-name	A symbol naming the COM implementation class for which this method is defined.
interface	A optional symbol which will be bound to the COM interface pointer whose method is being invoked. Usually this is not needed unless the interface pointer is being passed to some other function in the implementation.
arg-spec	Describes one of the method's arguments.
parameter-name	A symbol which will be bound to that argument's value while the *forms* are evaluated.

direction	Specifies the direction of the argument, either **`:in`**, **`:out`** or **`:in-out`** If specified, it must match the definition of the interface. The default is taken from the definition of the interface.
pass-style	Specifies how the argument will be converted to a Lisp value. It can be either **`:lisp`** or **`:foreign`**, the default is **`:lisp`**.
form	Forms which implement the method. The value of the final form is returned as the result of the method.

Description

The macro **`define-com-method`** defines a COM method that implements the method *method-name* for the COM implementation class *class-name*. The extended *method-spec* syntax is required if *class-name* implements more than one interface with a method called *method-name* (analogous to the C++ syntax **`InterfaceName::MethodName`**).

The symbol *this* is bound to the instance of the COM implementation class on which the method is being invoked. The symbol *this* is also defined as a local macro (as if by **`with-com-object`**), which allows the body to invoke other methods on the instance.

If present, the symbol *interface* is bound to the interface pointer on which the method is being invoked.

Each foreign argument is converted to a Lisp argument as specified by the *pass-style*. See Section 1.8.6, “Data conversion in define-com-method” for details.

If an error is to be returned from an Automation method, the function **`set-error-info`** can be used to provide more details to the caller.

Example

```
(define-com-method (i-robot rotate)
    ((this i-robot-impl)
     (axis :in)
     (angle-delta :in))
  (let ((joint (find-joint axis)))
    (rotate-joint joint))
  S_OK)
```

See also

define-com-implementation
set-error-info
set-variant

find-clsid *Function*

Summary

Searches the registry for a GUID or ProgId.

Package

com

Signature

find-clsid *name* &optional *errorp* => *refguid*

Arguments

name	A string or a **refguid**.
errorp	A generalized boolean.

Values

refguid	A **refguid**.

Description

The function **find-clsid** searches for the supplied GUID or ProgId in the registry.

name can be a string representing a GUID (with or without the curly brackets) or a string containing a ProgId. Otherwise *name* can be a **refguid**, which is simply returned.

If **find-clsid** fails to find the GUID, it either signals an error or returns **nil**, depending on the value of *errorp*. The default value of *errorp* is **t**.

Example

To find the GUID of the Explorer ActiveX:

```
(com:find-clsid "Shell.Explorer")
```

guid-equal *Function*

Summary Compares the GUID data in two GUID pointers.

Package `com`

Signature **`guid-equal`** ***guid1 guid2*** **`=>`** ***flag***

Arguments

guid1 A foreign pointer to a GUID object.

guid2 A foreign pointer to a GUID object.

Values

flag A boolean, true if *guid1* and *guid2* contain the same GUID data.

Description The function `guid-equal` compares the GUID data in *guid1* and *guid2* and returns true if the data is identical.

Examples

```
(guid-equal (com-interface-refguid 'i-unknown)
            (com-interface-refguid 'i-dispatch))
=> nil
```

```
(guid-equal (com-interface-refguid 'i-unknown)
            (make-guid-from-string
             "00000000-0000-0000-C000-000000000046"))
=> t
```

See also

`refguid`
`com-interface-refguid`
`guid-to-string`
`make-guid-from-string`
`refguid-interface-name`

guid-to-string *Function*

Summary Converts a GUID to a string of hex characters.

Package `com`

Signature **guid-to-string** ***guid*** **=>** ***guid-string***

Arguments *guid* A foreign pointer to a GUID object.

Values *guid-string* A string in the standard hex format for GUIDs.

Description The function **guid-to-string** converts the data in the *guid* to a string of hex characters in the standard-format.

Example

```
(guid-to-string (com-interface-refguid 'i-unknown))
=> "00000000-0000-0000-C000-000000000046"
```

See also **refguid**
com-interface-refguid
guid-equal
make-guid-from-string
refguid-interface-name

hresult *FLI type descriptor*

Summary The FLI type corresponding to **HRESULT** in C/C++.

Package **com**

Signature **hresult**

Description The **hresult** type is a signed 32 bit integer. When used as the result type of a COM method, the value **E_UNEXPECTED** is returned if the COM method body does not return an integer.

See also **hresult-equal**
check-hresult

hresult-equal *Function*

Summary Compares one **hresult** to another.

Package **com**

Signature **hresult-equal** ***hres1 hres2*** **=>** ***flag***

Arguments

hres1 An integer **hresult**.

hres2 An integer **hresult**.

Values

flag A boolean, true if *hres1* and *hres2* are equal.

Description The function **hresult-equal** compares *hres1* and *hres2* and returns true if they represent the same **hresult**. This function differs from the Common Lisp function **eql** because it handles signed and unsigned versions of each **hresult**.

Example **E_NOTIMPL** is negative, so

```
(eql E_NOTIMPL 2147500033)
=> nil
```

```
(hresult-equal E_NOTIMPL 2147500033)
=> t
```

See also

hresult

check-hresult

com-error

i-unknown *COM Interface Type*

Summary The Lisp name for the **IUnknown** COM interface.

Package **com**

Description The symbol **`i-unknown`** is the name given to the **`IUnknown`** COM interface within Lisp. The name results from the standard mapping described in Section 1.3, "The mapping from COM names to Lisp symbols".

Examples

```
(query-interface ptr 'i-unknown)
```

See also **`standard-i-unknown`**
`i-dispatch`

interface-ref *Macro*

Summary Accesses a place containing an interface pointer, maintaining reference counts.

Package **`com`**

Signature **`interface-ref`** ***iptr*** **`=>`** ***iptr***

`(setf interface-ref)` ***new-value*** ***iptr*** **`=>`** ***new-value***

Arguments

iptr	A place containing a COM interface pointer or **`nil`**.
new-value	A COM interface pointer or **`nil`**.

Description **`interface-ref`** is useful when manipulating a place containing an interface pointer.

The **`setf`** expander increments the reference count, as if by **`add-ref`**, of *new-value*, unless it is **`nil`**. It then decrements the reference count, as if by **`release`**, of the existing value in *iptr*, unless this is **`nil`**. Note that this order is important in the case that the new value is the same as the current value. Finally the value of place *iptr* is set to *new-value*.

The reader **`interface-ref`** simply returns its argument and does no reference counting. It may be useful in a form which both reads and writes a place like **`incf`**.

See also

`add-ref`
`release`

make-factory-entry *Function*

Summary
Make a object which can be used to register a class factory.

Package
`com`

Signature
`make-factory-entry &key` ***clsid implementation-name constructor-function constructor-extra-args friendly-name prog-id version-independent-prog-id***

Arguments

clsid The CLSID of the coclass.

implementation-name
A Lisp symbol naming the implementation class.

constructor-function
A function to construct the object. If **`nil`**, the default constructor is used which makes an instance of the *implementation-name* and queries it for a **`i-unknown`** interface pointer. The default constructor also handles *aggregation*.

constructor-extra-args
Extra arguments to pass to the *constructor-function*.

friendly-name The name of the coclass for use by application builders.

prog-id The ProgID of the coclass.

version-independent-prog-id

The VersionIndependentProgID of the coclass.

Description

Makes an object to contain all the information for class factory registration in the COM runtime. This object should be passed to **register-class-factory-entry** to perform the registration. This done automatically if you use **define-automation-component** described in the Chapter 3, "Using Automation".

Examples

```
(make-factory-entry
 :clsid (make-guid-from-string
         "7D9EB762-E4E5-11D5-BF02-000347024BE1")
 :implementation-name 'doc-impl
 :prog-id "Wordifier.Document.1"
 :version-independent-prog-id "Wordifier.Document"
 :friendly-name "Wordifier Document")
```

See also

register-class-factory-entry

make-guid-from-string *Function*

Summary

Make a **refguid** object from a hex string.

Package

com

Signature

make-guid-from-string ***string*** **&optional** ***interface-name***
=> ***refguid***

Arguments

string A string in the standard hex format for GUIDs.

interface-name A symbol naming a COM interface. If non-nil, the *refguid* will be will added to the table of known **refguids**.

Values

refguid A **refguid** object matching *string*.

Description The function **make-guid-from-string** makes a **refguid** object from *string*. If the GUID data matches a known **refguid**, then that is returned. Otherwise, a new **refguid** is created and returned. If *interface-name* is non-nil, then the table of known **refguid**s is updated. If the GUID is already known under a different name, an error is signalled.

Examples This GUID is a predefined one for **i-unknown**:

```
(refguid-interface-name
 (make-guid-from-string
  "00000000-0000-0000-C000-000000000046"))
=> I-UNKNOWN
```

See also **refguid**
com-interface-refguid
guid-equal
guid-to-string
refguid-interface-name

midl *Function*

Summary Converts an IDL file into Lisp FLI definitions.

Package **com**

Signature **midl** ***file*** **&key** ***package depth mapping-options output-file load import-search-path***

Arguments

file A pathname designator giving the name of an IDL file.

package The package in which definitions are created. Defaults to the current package.

depth How many levels of IDL **import** statement to convert to Lisp. This defaults to 0, which means only convert definitions for the IDL file itself. Imported files should be converted and loaded before the importing file. Some of the standard files are preloaded, so should not be loaded again (see Section 1.2.3, “Standard IDL files”).

mapping-options Allows options to be passed controlling the conversion of individual definitions.

output-file If this is **`nil`** (the default), the IDL file is compiled in-memory. Otherwise a Lisp fasl is produced so the definitions can be reloaded without requiring recompilation. If *output-file* is **`t`** then the fasl is named after the IDL file, otherwise *output-file* is used as a pathname designator to specify the name of the fasl file.

load If this is true (the default) then any fasl produced is loaded after being compiled. Otherwise, the fasl must be loaded explicitly with **`load`**. This argument has no effect if *output-file* is **`nil`**.

import-search-path

Specifies where to look for files referenced by **`import`** statements in the IDL. The default value, which is **`:default`**, causes a search in the same directory as *file*. Otherwise the value should be a list of pathname designators specifying directories to search. After searching using the value of *import-search-path*, **`midl`** looks in any directory in the **`INCLUDE`** environment variable.

Description This function is used to convert an IDL file into Lisp FLI definitions, which is necessary before the types in the file can be used from the Lisp COM API. See Section 1.3, “The mapping from COM names to Lisp symbols” for the details on how these FLI definitions are named.

Note: `midl` requires that types like `IDispatch` are declared before they are used.

Examples

To compile `myfile.idl` into memory:

```
(midl "myfile.idl")
```

To compile `myfile.idl` to `myfile.ofasl`:

```
(midl "myfile.idl" :output-file t :load nil)
```

To compile `myfile.idl` to `myfile.ofasl` and load it:

```
(midl "myfile.idl" :output-file t)
```

See also

`:midl-file`

:midl-file

Defsystem Member Type

Summary

The `:midl-file` defsystem member type can be used to include IDL files in a Lisp system definition.

Package

`com`

Description

When a file is given the type `:midl-file`, compiling the system will compile the IDL file to produce a fasl. Loading the system will load this fasl. The `:package`, `:mapping-options` and `:import-search-path` keywords can specified as for `midl`.

Examples

```
;; Include the file myfile.idl in a system
(defsystem my-system ()
  :members (("myfile.idl" :type :midl-file)))
```

See also

`midl`

query-interface *Function*

Summary Attempts to obtain a COM interface pointer for one interface from another.

Package `com`

Signature `query-interface` ***interface-ptr iid*** `&key` ***errorp*** `=>` ***interface-for-iid***

Arguments

interface-ptr	A COM interface pointer to be queried.
iid	The iid of a COM interface.
errorp	A boolean. The default is `t`.

Values

interface-for-iid	The new COM interface pointer or `nil`.

Description The function `query-interface` function invokes the COM method `IUnknown::QueryInterface` to attempt to obtain an interface pointer for the given *iid*. The *iid* can be a symbol naming a COM interface or a `refguid` foreign pointer containing its iid.

If the `IUnknown::QueryInterface` returns successfully then the new interface pointer *interface-for-iid* is returned.

If *errorp* is true, then `nil` is returned if the interface pointer cannot be found, otherwise an error of type `com-error` is signalled.

Example

```
(query-interface p-foo 'i-bar)
```

See also

`refguid`
`com-error`
`add-ref`
`release`
`with-temp-interface`
`with-query-interface`

query-object-interface *Macro*

Summary
Obtains a COM interface pointer for a particular interface from a COM object.

Package
com

Signature
query-object-interface ***class-name object iid*** **&key** ***ppv-object***
=> ***hresult*, *interface-ptr-for-iid***

Arguments

class-name	The COM object class name of the *object*. This can be a superclass name.
object	A COM object to be queried.
iid	The iid of a COM interface.
ppv-object	If specified, this should be a foreign pointer which will be set to contain the *interface-ptr-for-iid*.

Values

hresult	The **hresult**.
interface-ptr-for-iid	The new interface pointer or **nil** if none.

Description
The macro **query-object-interface** invokes the COM method **IUnknown::QueryInterface** to attempt to obtain an interface pointer for the given *iid*. The *iid* can be a symbol naming a COM interface or a **refguid** foreign pointer containing its iid.

The first value is the integer **hresult** from the call to **IUnknown::QueryInterface.** If the result indicates success, then *interface-ptr-for-iid* is returned as the second value.

Example
```
(query-object-interface foo-impl p-foo 'i-bar)
```

See also
refguid
hresult

refguid *FLI type descriptor*

Summary A FLI type used to refer to GUID objects.

Package `com`

Signature `refguid`

Description The **`refguid`** type is a pointer to a GUID structure, like the type **`REFGUID`** in C. In addition, a table of named **`refguid`**s is maintained, using the names chosen when COM interface types are converted to a Lisp FLI definitions by **`midl`** or parsing a type library.

Example

```
(typep (com-interface-refguid 'i-unknown) 'refguid)
=> t
```

See also
`com-interface-refguid`
`guid-equal`
`guid-to-string`
`make-guid-from-string`
`refguid-interface-name`
`refiid`
`midl`

refguid-interface-name *Function*

Summary Returns the COM interface name of a **`refguid`** if known.

Package `com`

Signature **`refguid-interface-name`** ***refguid*** **`=>`** ***interface-name***

Arguments *refguid* A **`refguid`** object.

Values *interface-name* A symbol naming the COM interface of *refguid*.

Description
: Returns a symbol naming the COM interface of *refguid*, which must be a **refguid** object known to Lisp.

Example

```
(refguid-interface-name
 (make-guid-from-string
  "00000000-0000-0000-C000-000000000046"))
=> i-unknown
```

See also
: **refguid**
com-interface-refguid
guid-equal
guid-to-string
make-guid-from-string

refiid *FLI type descriptor*

Summary
: A FLI type used to refer to iids.

Package
: **com**

Signature
: **refiid**

Description
: The **refgiid** foreign type is a useful converted type for iid arguments to foreign functions. When given a symbol, it looks up the GUID as if by calling **com-interface-refguid**. Otherwise the value should be a foreign pointer to a GUID structure, which is passed directly without conversion.

Example
: Given the definition of **print-iid**:

```
(fli:define-foreign-function print-iid
    ((iid refiid)))
```

then these two forms are equivalent:

```
(print-iid 'i-unknown)
```

```
(print-iid (com-interface-refguid 'i-unknown))
```

See also **com-interface-refguid**
refguid

register-class-factory-entry *Function*

Summary Registers the description of a class factory.

Package **com**

Signature **register-class-factory-entry** ***new-factory-entry***

Arguments *new-factory-entry*

A factory entry from **make-factory-entry**.

Description Register the factory entry with the COM runtime so that **register-server**, **unregister-server**, **start-factories** and **stop-factories** will know about the coclass in the factory entry. This is done automatically if you use **define-automation-component** described in the Chapter 3, “Using Automation”.

Examples

See also **make-factory-entry**
start-factories
stop-factories
register-server
unregister-server

register-server *Function*

Summary Externally registers all class factories known to Lisp.

Package **com**

Signature **register-server &key *clsctx***

Arguments

clsctx The CLSCTX in which to register the class factory.

Description The **register-server** function updates the Windows registry to contain the appropriate keys for all the class factories registered in the current Lisp image. For Automation components, the type libraries are registered as well. During development, the type library will be found whereever the system definition specified, but after using Lispworks delivery it must be located in the directory containing the application's executable or DLL.

This function should be called when an application is installed, usually by detecting the **/RegServer** command line argument.

When running on 64-bit Windows, 32-bit LispWorks updates the 32-bit registry view and 64-bit LispWorks updates the 64-bit registry view. LispWorks does not change the registry reflection settings.

Example

```
(defun start-up-function ()
  (cond ((member "/RegServer"
                 system:*line-arguments-list*
                 :test 'equalp)
         (register-server))
        ((member "/UnRegServer"
                 system:*line-arguments-list*
                 :test 'equalp)
         (unregister-server))
        (t
         (co-initialize)
         (start-factories)
         (start-application-main-loop)))
  (quit))
```

See also **unregister-server**

register-class-factory-entry

start-factories
stop-factories

release *Function*

Summary
: The **release** function decrements the reference count of an interface pointer.

Package
: **com**

Signature
: **release** ***interface-ptr*** **=>** ***ref-count***

Arguments
: *interface-ptr* A COM interface pointer.

Values
: *ref-count* The new reference count.

Description
: Each COM interface pointer has a reference count which is used by the server to control its lifetime. The function **release** should be called whenever a reference to the interface pointer is being removed. The function invokes the COM method **IUnknown::Release** so the form **(release ptr)** is equivalent to using **call-com-interface** as follows:

```
(call-com-interface (ptr i-unknown release))
```

Example

```
(release p-foo)
```

See also
: **add-ref**
interface-ref
query-interface
with-temp-interface

s_ok *Macro*

Summary
: Compares a result code to the value of **s_ok**.

Package **com**

Signature **s_ok** ***hresult*** **=>** ***flag***

Arguments *hresult* An integer **hresult**.

Values *flag* A boolean.

Description The **s_ok** macro checks the *hresult* and returns true if its value is that of the constant **S_OK**.

Examples

```
(S_OK S_OK) => t

(S_OK S_FALSE) => nil

(S_OK E_NOINTERFACE) => nil
```

See also
succeeded
hresult
hresult-equal
check-hresult

standard-i-unknown *Class*

Summary A complete implementation of the **i-unknown** interface.

Package **com**

Superclasses **com-object**

Subclasses
standard-i-dispatch
standard-i-connection-point-container

Initargs **:outer-unknown**

An optional interface pointer to the outer unknown interface if this object is *aggregated*.

Description

The class **standard-i-unknown** provides a complete implementation of the **i-unknown** interface.

The class provides a reference count for the object which calls the generic function **com-object-initialize** when the object is given a reference count and **com-object-destructor** when it becomes zero again. These generic functions can be specialized to perform initialization and cleanup operations.

The class also provides an implementation of **query-interface** which calls the generic function **com-object-query-interface**. The default method handles **i-unknown** and all the interfaces specified by the **define-com-implementation** form for the class of the object.

There is support for *aggregation* via the **:outer-unknown** initarg, which is also passed by built-in class factory implementation.

Example

Inheriting from a non-COM class requires **standard-i-unknown** to be mentioned explicitly:

```
(define-com-implementation doc-impl
                           (document-mixin
                            standard-i-unknown)
  ()
  (:interfaces i-doc))
```

See also

define-com-implementation
standard-i-dispatch
standard-i-connection-point-container
com-object-initialize
com-object-destructor
com-object-query-interface
com-object
i-unknown

start-factories *Function*

Summary Starts all the registered class factories.

Package **`com`**

Signature **`start-factories &optional`** ***clsctx***

Arguments *clsctx* The CLSCTX in which to start the factories.

Description The **`start-factories`** function starts all the registered class factories in the given *clsctx*, which defaults to **`CLSCTX_LOCAL_SERVER`**. This function should be called once when a COM server application starts if it has externally registered class factories.

See also **`register-class-factory-entry`**
`stop-factories`
`register-server`
`unregister-server`
`co-initialize`

stop-factories *Function*

Summary Stops all the registered class factories.

Package **`com`**

Signature **`stop-factories`**

Description The **`stop-factories`** function stops all the registered class factories. This function should be called once before a COM server application exits if it has externally registered class factories.

See also **register-class-factory-entry**
start-factories
register-server
unregister-server
co-uninitialize

succeeded *Macro*

Summary Checks an **hresult** for success.

Package **com**

Signature **succeeded *hresult* => *flag***

Arguments *hresult* An integer **hresult**.

Values *flag* A boolean.

Description The **succeeded** macro checks the *hresult* and returns true if the it is one of the 'succeeded' values, for instance **S_OK** or **S_FALSE**.

Examples

```
(succeeded S_OK) => t

(succeeded E_NOINTERFACE) => nil
```

See also **check-hresult**
hresult
hresult-equal
s_ok

unregister-server *Function*

Summary Externally unregisters all class factories known to Lisp.

Package **com**

Signature `unregister-server`

Description The `unregister-server` function updates the Windows registry to remove the appropriate keys for all the class factories registered in the current Lisp image. For Automation components, the type libraries are unregistered as well.

This function should be called when an application is uninstalled, usually by detecting the `/UnRegServer` command line argument.

When running on 64-bit Windows, 32-bit LispWorks updates the 32-bit registry view and 64-bit LispWorks updates the 64-bit registry view. LispWorks does not change the registry reflection settings.

Example

```
(defun start-up-function ()
  (cond ((member "/UnRegServer"
                 system:*line-arguments-list*
                 :test 'equalp)
         (unregister-server))
        ((member "/RegServer"
                 system:*line-arguments-list*
                 :test 'equalp)
         (register-server))
        (t
         (co-initialize)
         (start-factories)
         (start-application-main-loop)))
  (quit))
```

See also

`register-server`
`register-class-factory-entry`
`start-factories`
`stop-factories`

with-com-interface *Macro*

Summary Used to simplify invocation of several methods from a particular COM interface pointer.

Package `com`

Signature **`with-com-interface`** ***disp interface-ptr form**** **`=>`** ***values***

disp **`::=`** **`(`*****dispatch-function interface-name*****`)`**

Arguments

disp	The names of the dispatch function and interface.
dispatch-function	A symbol which will be defined as a local macro, as if by **`macrolet`**. The macro can be used by the *form*s to invoke the methods on *interface-ptr*.
interface-name	A symbol which names the COM interface. It is not evaluated.
interface-ptr	A form which is evaluated to yield a COM interface pointer that implements *interface-name*.
form	A form to be evaluated.

Values

values	The values returned by the last *form*.

Description When the macro **`with-com-interface`** evaluates the *form*s, the local macro *dispatch-function* can be used to invoked the methods for the COM interface *interface-name*, which should be the type or a supertype of the actual type of *interface-ptr*.

The *dispatch-function* macro has the following signature:

dispatch-function method-name arg* **`=>`** ***values***

where

method-name	A symbol which names the method. It is not evaluated.

arg — Arguments to the method (see Section 1.7.1, "Data conversion when calling COM methods" for details).

values — Values from the method (see Section 1.7.1, "Data conversion when calling COM methods" for details).

Example This example invokes the COM method **`GetTypeInfo`** in the interface **`IDispatch`**.

```
(defun get-type-info (disp tinfo &key
                         (locale LOCALE_SYSTEM_DEFAULT))
  (multiple-value-bind (hres typeinfo)
      (with-com-interface (call-disp i-dispatch) disp
        (call-disp get-type-info tinfo locale))
    (check-hresult hres 'get-type-info)
    typeinfo))
```

See also **`call-com-interface`**

with-com-object *Macro*

Summary Used to simplify invocation of several methods from a given COM object.

Package **`com`**

Signature **`with-com-object`** ***disp object form**** **`=>`** ***values***

disp **`::=`** **`(`*****dispatch-function class-name*** **`&key`** ***interface*****`)`**

Arguments

disp — The names of the dispatch function and object class.

dispatch-function — A symbol which will be defined as a macro, as if by **`macrolet`**. The macro can be used by the *form*s to invoke the methods on *object*.

	class-name	A symbol which names the COM implementation class. It is not evaluated.
	interface	An optional form which when evaluated should yield a COM interface pointer. This is only needed if the definition of the methods being called have the *interface* keyword in their *class-spec*s.
	object	A form which is evaluated to yield a COM object.
	form	A form to be evaluated.
Values	*values*	The values returned by the last *form*.

Description

When the macro **`call-com-object`** evaluates the *form*s, the local macro *dispatch-function* can be used to invoked the methods for the COM class *class-name*, which should be the type or a supertype of the actual type of *object*.

The *dispatch-function* macro has the following signature:

dispatch-function method-spec arg* => values

method-spec ::= method-name | (interface-name method-name)

where

method-spec	Specifies the method to be called. It is not evaluated.
method-name	A symbol naming the method to call.
interface-name	A symbol naming the interface of the method to call. This is only required if the implementation class *class-name* has more than one method with the given *method-name*.
arg	Arguments to the method (see Section 1.9.1, “Data conversion when calling COM object methods” for details).

values — Values from the method (see Section 1.9.1, "Data conversion when calling COM object methods" for details).

Note that, because **`with-com-object`** requires a COM object, it can only be used by the implementation of that object. All other code should use **`with-com-interface`** with the appropriate COM interface pointer.

Example

```
(with-com-object (call-my-doc doc-impl) my-doc
  (call-my-doc move 0 0)
  (call-my-doc resize 100 200))
```

See also

`call-com-object`
`define-com-method`
`with-com-interface`

with-temp-interface *Macro*

Summary — Used to simplify reference counting for a COM interface pointer.

Package — **`com`**

Signature — **`with-temp-interface`** **(*var*)** ***interface-ptr form**** **`=>`** ***values***

Arguments

var — A variable which is bound to *interface-ptr* while the *form*s are evaluated.

interface-ptr — A form which is evaluated to yield a COM interface pointer.

form — A form to be evaluated.

Values

values — The values returned by the last *form*.

Description

When the macro **`with-temp-interface`** evaluates the *form*s, the variable *var* is bound to the value of *interface-ptr*. When control leaves the body (whether directly or due to a non-local exit), **release** is called with this interface pointer.

Example

This example invokes the COM method **GetDocumentation** in the interface **ITypeInfo** on an interface pointer which must be released after use.

```
(defun get-tinfo-member-documentation (disp tinfo
                                            member-id)
  (with-temp-interface (typeinfo)
      (get-type-info disp tinfo)
    (call-com-interface (typeinfo i-type-info
                                  get-documentation)
                        member-id)))
```

See also

release
with-query-interface

with-query-interface *Macro*

Summary

Used to simplify reference counting when querying a COM interface pointer.

Package

com

Signature

with-query-interface ***disp interface-ptr form**** **=>** ***values***

disp **::=** **(*****punknown interface-name*** **&key** ***errorp dispatch*****)**

Arguments

punknown — A variable which is bound to the queried interface pointer while the *form*s are evaluated.

interface-name — A symbol which names the COM interface. It is not evaluated.

errorp — A boolean indicating whether an error should be signaled if *interface-name* is not implemented by *interface-ptr*.

dispatch	A symbol which will be defined as a local macro, as if by **macrolet** as if by **with-com-interface**. The macro can be used by the *form*s to invoke the methods on *punknown*.
interface-ptr	A form which is evaluated to yield a COM interface pointer to query.
form	A form to be evaluated.

Values *values* The values returned by the last *form*.

Description

The macro **with-query-interface** calls **query-interface** to find an interface pointer for *interface-name* from the existing COM interface pointer *interface-ptr*. While evaluates the *form*s, the variable *punknown* is bound to the queried pointer and the pointer is released when control leaves the body (whether directly or due to a non-local exit).

If *errorp* is true, then *punknown* is bound to **nil** if the interface pointer cannot be found, otherwise an error of type **com-error** is signalled.

If *dispatch* is specified, then a local macro is created as if by **with-com-interface** to invoke COM interface methods on *punknown*.

Example

This example invokes the methods on an **i-bar** interface pointer queried from an existing interface pointer.

```
(with-query-interface (p-bar i-bar
                       :dispatch call-bar)
    p-foo
  (call-bar bar-init)
  (call-bar bar-print))
```

See also

query-interface
release
with-temp-interface

3

Using Automation

3.1 Including Automation in a Lisp application

3.1.1 Loading the modules

Before using any of the LispWorks Automation APIs, you need to load the module using

```
(require "automation")
```

3.1.2 Generating FLI definitions from COM definitions

Automation components and interfaces that are to be used by the Automation API must be placed in a type library using suitable tools. In some cases, this type library will be supplied as part of the DLL or executable containing the component.

Some of the Automation APIs described in this chapter require you to convert the definitions in the type library into FLI definitions. This is done by compiling and load a system definition that references the library with the options `:type :midl-type-library-file`. The names in the type library are converted to Lisp symbols as specified in “The mapping from COM names to Lisp symbols” on page 3

Note: this is not required by all the APIs, for example see “Calling Automation methods without a type library” on page 80 and “A simple implementation of a single Automation interface” on page 84.

3.1.3 Reducing the size of the converted library

Suppose you have a **`defsystem`** system definition form that references a library: that is, a system member has options **`:type :midl-type-library-file`** as described in “Generating FLI definitions from COM definitions” on page 77.

For this member, the option **`:com`** can be added to specify whether all the COM functionality is required. The keyword can take these values:

`t`	Analyze and generate all the required code for calling and implementing the interfaces from the type library. This is the default value.
`nil`	Analyze but do not generate any code for calling or implementing COM interfaces from the type library. It is still possible to call Automation methods.
`:not-binary`	Analyze but do not generate any code for calling or implementing COM interfaces from the type library. It is still possible to call Automation methods and implement *dispinterfaces* in the type library, but not dual or COM interfaces.

Using the value **`nil`** or **`:not-binary`** generates much smaller code and is therefore much faster. However, it is never obligatory to use the option **`:com`**.

Use **`:com nil`** when the application calls Automation interfaces from the type library but does not implement any of them or need to call any methods from dual interfaces using **`call-com-interface`**.

Use **`:com :not-binary`** when the application implements only *dispinterfaces* from the library. This is typically required for implementing *sink* interfaces for use with connection points.

For an example see **`examples/com/ole/simple-container/defsys.lisp`**.

3.2 Starting a remote Automation server

A remote Automation server is started from Lisp by using its coclass name, CLSID or ProgID. The macro **`with-coclass`** can be used to make an instance of an automation server from its coclass name for the duration of its body. The function **`create-object`** can be used to start an automation server given its CLSID or ProgID. The function **`create-instance-with-events`** can be used to start and automation server and set its event handler.

3.3 Calling Automation methods

Automation methods can be called either with or without a compiled type library. In both cases, arguments and return values are converted according to the types specified by the method's definition.

3.3.1 Calling Automation methods using a type library

To use this approach, you must have the type library available at compile-time (see "Generating FLI definitions from COM definitions" on page 77). Information from the type library is built into your application, which makes method calling more efficient. However, it also makes it less dynamic, because the library at the time the application is run must match.

There are three kinds of Automation method, each of which is called using macros designed for the purpose.

- Ordinary method are called using the macros **`call-dispatch-method`** and **`with-dispatch-interface`**. If there is no Automation method with the given method name, then a property getter with the same name is called if it exists, otherwise an error is signaled. The **`setf`** form of **`call-dispatch-method`** can be used to call property setter methods.
- Property getter methods are called using the macro **`call-dispatch-get-property`**.
- Property setter methods are called using the macros **`call-dispatch-put-property`** or the **`setf`** form of **`call-dispatch-get-property`**.

To use these macros, you need to specify the interface name, the method name, a COM interface pointer for the **`i-dispatch`** interface and suitable arguments. The interface and method names are given as symbols named as in Section 1.3 on page 3 and the COM interface pointer is a foreign pointer of type **com-interface**. In all the macros, the *arg*s and *value*s are as specified in the Section 3.3.3.

The **with-dispatch-interface** macro is useful when several methods are being called with the same COM interface pointer, because it establishes a local macro that takes just the method name and arguments.

3.3.2 Calling Automation methods without a type library

This approach is useful if the type library is not available at compile time or you want to allow methods to be called dynamically without knowing the interface pointer type at compile-time. It can be less efficient than using the approach in Section 3.3.1, but is often the simplest approach, especially if the Automation component was written to be called from a language like Visual Basic.

There are three kinds of Automation method, each of which is called using functions designed for the purpose.

- Ordinary method are called using the function **invoke-dispatch-method**. If there is no Automation method with the given method name, then a property getter with the same name is called if it exists, otherwise an error is signaled. The **setf** form of **invoke-dispatch-method** can be used to call property setter methods.
- Property getter methods are called using the function **invoke-dispatch-get-property**.
- Property setter methods are called either using the function **invoke-dispatch-put-property** or the **setf** form of **invoke-dispatch-get-property**.

To use these function, you need to specify a COM interface pointer for the **i-dispatch** interface, the method name and suitable arguments. The method name is given as a string or integer and the COM interface pointer is a foreign pointer of type **com-interface**. In all the functions, the *arg*s and *value*s are as specified in the Section 3.3.3.

3.3.3 Data conversion when calling Automation methods

The arguments and return values to Automation methods are restricted to a small number of simple types, which map to Lisp types as follows:

Table 3.1 Automation types, VT codes and their corresponding Lisp types

Automation type	VT code	Lisp type
null value	**VT_NULL**	the symbol **:null**
empty value	**VT_EMPTY**	the symbol **:empty**
SHORT	**VT_I2**	**integer**
LONG	**VT_I4**	**integer**
FLOAT	**VT_R4**	**single-float**
DOUBLE	**VT_R8**	**double-float**
CY	**VT_CY**	not suppported
DATE	**VT_DATE**	not suppported
BSTR	**VT_BSTR**	**string**
IDispatch*	**VT_DISPATCH**	FLI **(:pointer i-dispatch)**
SCODE	**VT_ERROR**	**integer**
VARIANT_BOOL	**VT_BOOL**	**nil** or **t**
VARIANT*	**VT_VARIANT**	recursively convert
IUknown*	**VT_UNKNOWN**	FLI **(:pointer i-unknown)**
DECIMAL	**VT_DECIMAL**	not suppported
BYTE	**VT_UI1**	**integer**
SAFEARRAY	**VT_ARRAY**	**array**
dynamic	dynamic	**lisp-variant**

When an Automation argument is a **lisp-variant** object, its type is used to set the VT code. See **make-lisp-variant** and **set-variant**.

In and *in-out* parameters are passed as positional arguments in the calling forms and *out* and *in-out* parameters are returned as additional values. If there is an argument with the **retval** attribute then it is returned as the first value.

3.3.4 Using collections

The macro **do-collection-items** can be used to iterate over the items or an interface that implements the Collection protocol. If the collection items are interface pointers, they must be released when not needed.

For example, to iterate over the **Table** objects from the **Tables** collection of a **MyDocument** interface pointer

```
(with-temp-interface (tables)
    (call-dispatch-get-property
       (doc my-document tables))
  (do-collection-items (table tables)
    (inspect-the-table table)
    (release table)))
```

3.3.5 Using connection points

Event *sink* interfaces can be connected and disconnected using the functions **interface-connect** and **interface-disconnect**.

For example, the following macro connects a sink interface pointer *event-handler* to a source of **i-clonable-events** events *clonable* for the duration of its body.

```
(defmacro handling-clonable-events ((clonable event-handler)
                                    &body body)
  (lw:with-unique-names (cookie)
    (lw:rebinding (clonable event-handler)
      `(let ((,cookie nil))
         (unwind-protect
             (progn
               (setq ,cookie
                     (interface-connect ,clonable
                                        'i-clonable-events
                                        ,event-handler))
               ,@body)
           (when ,cookie
             (interface-disconnect ,clonable
                                   'i-clonable-events
                                   ,cookie)))))))
```

3.3.6 Error handling

When an Automation server returns an error code, the calling macros such as **call-dispatch-method** signal an error of type **com-error**. The error message will contain the *source* and *description* fields from the error.

For example, if **pp** is a dispatch pointer to **i-test-suite-1**:

```
CL-USER 184 > (call-dispatch-method
                (pp nil i-test-suite-1 fx))
"in fx"            ;; implementation running
Error: COM IDispatch::Invoke Exception Occured (0 "fx") : foo
  1 (abort) Return to level 0.
  2 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed,  or :? for
other options
```

3.4 Implementing Automation interfaces in Lisp

This section describes two techniques for implementing Automation interfaces in Lisp. The choice of technique usually depends on whether you are implementing a complete server or a simple event sink. The section then describes other kinds of interfaces that can be implemented and how to report errors to the caller of a method.

3.4.1 A complete implementation of an Automation server

In the case where you are designing an set of COM interfaces and implementing a server to support them, you need to make a complete implementation in Lisp. This allows several Automation interfaces to be implemented by a single class and also supports *dual* interfaces.

The implementation defines an appropriate class, inheriting from the class **standard-i-dispatch** to obtain an implementation of the COM interface **i-dispatch**. This implementation of **i-dispatch** will automatically invoke the appropriate COM method.

For *dual* interfaces, the methods should be defined in the same way as described for COM interfaces in Section 1.8 on page 13.

For *dispinterfaces*, the methods should be implemented using the macro **define-dispinterface-method** or by a specialized method of the generic function **com-object-dispinterface-invoke**.

To implement an Automation interface in Lisp with **standard-i-dispatch**, you need the following:

1. A type library for the component, converted to Lisp as specified in Section 3.1 on page 77.
2. A COM object class defined with **define-automation-component** or **define-automation-collection**, specifying the coclass or interface(s) to implement.
3. Implementations of the methods using **define-com-method**, **define-dispinterface-method** or **com-object-dispinterface-invoke**.
4. For an out-of-process Automation component, registration code which calls **register-server** and **unregister-server**, typically in response to the command line arguments **/RegServer** and **/UnRegServer**.
5. Initialization code which calls **co-initialize** and **start-factories** in a thread that will be processing Windows messages (for instance a CAPI thread).

3.4.2 A simple implementation of a single Automation interface

In the case where you are implementing a single dispinterface that was designed by someone else, for example an *event sink*, you can usually avoid needing to parse a type library or define a class to implement the interface.

Instead, you implement a dispinterface using the class **simple-i-dispatch** by doing the following:

1. Obtain an interface pointer that will provide type information for the component, to be used as the *related-dispatch* argument in the call to the function **query-simple-i-dispatch-interface**. In the case where you are implementing an event sink, the source interface pointer will usually do this.

2. Optionally, define a class with **defclass** inheriting from **simple-i-dispatch**. The class **simple-i-dispatch** can be used itself if no special callback object is required.
3. Implement an *invoke-callback* that selects and implements the methods of the interface.
4. Define initialization code which calls **co-initialize**, obtains the *related-dispatch* from step 1, makes an instance of the COM object class defined in step 2 with the *invoke-callback* from step 3, obtains its interface pointer by calling **query-simple-i-dispatch-interface** (passing the *related-dispatch*) and attaches this interface pointer to the appropriate sink in the *related-dispatch* (for example using connection point functions such as **interface-connect**). This must all be done in a thread that will be processing Windows messages (for instance a CAPI thread).

3.4.3 Implementing collections

Interfaces that support the Collection protocol can be implemented using the macro **define-automation-collection**. This defines a subclass of **standard-automation-collection**, which implements the minimal set of collection methods and calls Lisp functions to provide the items. If the collection items are interface pointers, appropriate reference counting must be observed.

See the example in the directory **examples/com/automation/collections/**.

3.4.4 Implementing connection points

Lisp implementations can act as *event sources* via a built-in implementation of the **IConnectionPointContainer** interface, which **define-automation-component** provides if *source* interfaces are specified. A built-in implementation of **IConnectionPoint** handles connections for each interface and the macro **do-connections** can be used to iterate over the connections when firing the events.

3.4.5 Reporting errors

Classes defined using **define-automation-component** allow extended error information to be returned for all Automation methods. Within the body of a `define-com-method` definition, the function `set-error-info` can be called to describe the error. In addition, this function returns the value of `DISP_E_EXCEPTION`, which can be returned directly as the `hresult` from the method.

For example:

```
(define-com-method (i-test-suite-1 fx)
    ((this c-test-suite-1))
  (print "in fx")
  (set-error-info :description "foo"
                  :iid 'i-test-suite-1
                  :source "fx"))
```

3.4.6 Automation of a CAPI application

For an example of how to implement an Automation server that controls a CAPI application, see the file `examples\com\automation\capi-application\build.lisp` in the LispWorks installation.

4

Automation Reference Entries

The following chapter documents Automation functionality.

call-dispatch-get-property *Macro*

Summary Calls an Automation property getter method from a particular interface.

Package **com**

Signature **call-dispatch-get-property** ***spec arg**** **=>** ***values***

spec **::=** **(*dispinterface-ptr dispinterface-name method-name*)**

Arguments

spec The interface pointer and a specification of the method to be called.

dispinterface-ptr A form which is evaluated to yield a COM **i-dispatch** interface pointer.

dispinterface-name

A symbol which names the Automation interface. It is not evaluated.

method-name	A symbol which names the property getter method. It is not evaluated.
arg	Arguments to the method (see Section 3.3.3, "Data conversion when calling Automation methods" for details).

Values

values	Values from the method (see Section 3.3.3, "Data conversion when calling Automation methods" for details).

Description

The **`call-dispatch-get-property`** macro is used to invoke an Automation property getter method from Lisp. The *dispinterface-ptr* should be a COM interface pointer for the **`i-dispatch`** interface. The appropriate Automation property getter method, chosen using *dispinterface-name* and *method-name*, is invoked after evaluating each *arg*. The *arg*s must be values that are suitable for the method and of types compatible with Automation. The values returned are as specified by the method signature. In general, property getter methods take no arguments and return the value of the property, but see Section 3.3.3, "Data conversion when calling Automation methods" for more details.

There is also **`setf`** expander for **`call-dispatch-get-property`**, which can be used as an alternative to the **`call-dispatch-put-property`** macro.

Example

For example, in order to get and set the **`Width`** property of a **`MyDocument`** interface pointer

```
(call-dispatch-get-property
   (doc my-document width))

(setf (call-dispatch-get-property
          (doc my-document width))
      10)
```

See also

`call-dispatch-put-property`
`call-dispatch-method`

call-dispatch-method *Macro*

Summary Calls an Automation method from a particular interface.

Package `com`

Signature `call-dispatch-method` ***spec arg**** `=>` ***values***

spec `::=` `(`***dispinterface-ptr dispinterface-name method-name***`)`

Arguments

spec The interface pointer and a specification of the method to be called.

dispinterface-ptr A form which is evaluated to yield a COM `i-dispatch` interface pointer.

dispinterface-name

A symbol which names the Automation interface. It is not evaluated.

method-name A symbol which names the method. It is not evaluated.

arg Arguments to the method (see Section 3.3.3, "Data conversion when calling Automation methods" for details).

Values

values Values from the method (see Section 3.3.3, "Data conversion when calling Automation methods" for details).

Description

The `call-dispatch-method` macro is used to invoke an Automation method from Lisp. The *dispinterface-ptr* should be a COM interface pointer for the `i-dispatch` interface. The appropriate Automation method, chosen using *dispinterface-name* and *method-name*, is invoked after evaluating each *arg*. The *arg*s must be values that are suitable for the method and of types compatible with Automation. The values returned are as specified by the method signature. See Section 3.3.3, "Data conversion when calling Automation methods" for more details. If there is no Automation method with the given *method-name*, then a property getter with the same name is called if it exists, otherwise an error is signaled. The **setf** form of **call-dispatch-method** can be used to call property setter methods.

Example

For example, in order to invoke the **ReFormat** method of a **MyDocument** interface pointer

```
(call-dispatch-method (doc my-document re-format))
```

See also

with-dispatch-interface
call-dispatch-get-property
call-dispatch-put-property

call-dispatch-put-property *Macro*

Summary

Calls an Automation property setter method from a particular interface.

Package

com

Signature

call-dispatch-put-property ***spec arg**** **=>** ***values***

spec **::=** **(*dispinterface-ptr dispinterface-name method-name*)**

Arguments

spec	The interface pointer and a specification of the method to be called.
dispinterface-ptr	A form which is evaluated to yield a COM **i-dispatch** interface pointer.
dispinterface-name	A symbol which names the Automation interface. It is not evaluated.
method-name	A symbol which names the property getter method. It is not evaluated.

	arg	Arguments to the method (see Section 3.3.3, "Data conversion when calling Automation methods" for details).
Values	*values*	Values from the method (see Section 3.3.3, "Data conversion when calling Automation methods" for details).

Description

The **call-dispatch-put-property** macro is used to invoke an Automation property setter method from Lisp. The *dispatch-ptr* should be a COM interface pointer for the **i-dispatch** interface. The appropriate Automation property setter method, chosen using *dispinterface-name* and *method-name*, is invoked after evaluating each *arg*. The *arg*s must be values that are suitable for the method and of types compatible with Automation. The values returned are as specified by the method signature. In general, property setter methods take one argument (the new value) and return the no values, but see Section 3.3.3, "Data conversion when calling Automation methods" for more details.

There is also **setf** expander for **call-dispatch-get-property**, which can be used as an alternative to the **call-dispatch-put-property** macro.

Example

For example, in order to set the **Width** property of a **MyDocument** interface pointer

```
(call-dispatch-put-property
   (doc my-document width)
   10)
```

See also

call-dispatch-get-property
call-dispatch-method

com-object-dispinterface-invoke *Generic Function*

Summary A generic function called by `IDispatch::Invoke` when there is no defined *dispinterface* method.

Package `com`

Signature `com-object-dispinterface-invoke` ***com-object method-name method-type args*** `=>` ***value***

Arguments

com-object	A COM object whose method is being invoked.
method-name	A string naming the method to be called.
method-type	A keyword specifying the type of method being called.
args	A vector containing the arguments to the method.

Description The generic function `com-object-dispinterface-invoke` is called by `IDispatch::Invoke` when there is no method defined using `define-dispinterface-method`.

Methods can be written for `com-object-dispinterface-invoke`, specializing on an Automation implementation class and implementing the method dispatch based on *method-name* and *method-type*.

The *method-name* argument is a string specifying the name of the method as given by the method declaration in the IDL or type library. The *method-type* argument, has one of the following values:

`:get`	when invoking a property getter method.
`:put`	when invoking a property setter method.
`:method`	when invoking a normal method.

The arguments to the method are contained in the vector *args*, in the order specified by the method declaration in the type library. For *in* and *in-out* arguments, the corresponding element of *args* contains the argument value converted to the type specified by the method declaration and then converted to Lisp objects as specified in Section 3.3.3, "Data conversion when calling Automation methods". For *out* and *in-out* arguments, the corresponding element of *args* should be set by the method to contain the value to be returned to the caller and will be converted to an automation value as specified in Section 3.3.3, "Data conversion when calling Automation methods".

The *value* should be a value which can be converted to the appropriate return type as the primary value of the method and will be converted to an automation value as specified in Section 3.3.3, "Data conversion when calling Automation methods". It is ignored for methods that are declared as returning void.

Note: when using **com-object-dispinterface-invoke**, it is not possible to distinguish between invocations of the same method name for different interfaces when *com-object* implements several interfaces. If this is required, then the method must be defined with **define-dispinterface-method**.

Example

See the example file in

examples/com/ole/simple-container/owc-spread-sheet.lisp

See also

define-dispinterface-method

create-instance-with-events *Function*

Summary

A convenience function which combines **create-instance** and **set-i-dispatch-event-handler**.

Package **`com`**

Signature **`create-instance-with-events`** ***clsid event-handler*** **`&rest`** ***args***
`&key` ***event-object*** **`=>`** ***interface*, *list***

Arguments *clsid* A string or a **`refguid`** giving a CLSID to create.

event-handler A function of four arguments.

event-object A Lisp object.

Values *interface* An **`i-dispatch`** interface.

sinks A list of objects representing the connections made.

Description The function **`create-instance-with-events`** is a convenience function which starts an **`i-dispatch`** interface and sets an event handler.

It first calls **`create-instance`** with *clsid* and all the keyword arguments except the *event-object*. It defaults the **`create-instance`** argument *riid* to the value **`i-dispatch`**.

It then calls **`set-i-dispatch-event-handler`** on the resulting interface, passing *event-handler*, *event-object* and *clsid* (as the *coclass*).

interface is the interface started, and *sinks* is the result of **`set-i-dispatch-event-handler`**.

Examples See **`examples/com/automation/events/ie-events.lisp`**

See also **`create-instance`**
`set-i-dispatch-event-handler`

create-object *Function*

Summary Create an instance of a coclass.

Package `com`

Signature **`create-object &key`** ***clsid progid clsctx*** **`=>`** ***interface-ptr***

Arguments

clsid A string giving a CLSID to create.

progid A string giving a ProgID to create.

clsctx A CLSCTX value, which defaults to **`CLSCTX_SERVER`**.

Values

interface-ptr An **`i-dispatch`** interface pointer.

Description Creates an instance of a coclass and returns its **`i-dispatch`** interface pointer. The coclass can be specified directly by using the *clsid* argument or indirectly using the *progid* argument, which will locate the CLSID from the registry.

Examples The following are equivalent ways of creating an Microsoft Word application object:

```
(create-object :progid "Word.Application.8")
```

```
(create-object
  :clsid "000209FF-0000-0000-C000-000000000046")
```

See also **`with-coclass`**

define-automation-collection *Macro*

Summary Defines an implementation class for an Automation component that supports the Collection protocol.

Package `com`

Signature **`define-automation-collection`** ***class-name*** **(*superclass-name**)** **(*slot-specifier**)** ***class-option****

Arguments

class-name A symbol naming the class to define.

superclass-name	A symbol naming a superclass to inherit from.
slot-specifier	A slot description as used by **`defclass`**.
class-option	An option as used by **`defclass`**.

Description

The macro **`define-automation-collection`** defines a **`standard-class`** which is used to implement an Automation component that supports the Collection protocol. Normal **`defclass`** inheritance rules apply for slots and Lisp methods.

Each *superclass-name* argument specifies a direct superclass of the new class, which can be any **`standard-class`** provided that **`standard-automation-collection`** is included somewhere in the overall class precedence list. This standard class provides a framework for the collection class.

*slot-specifier*s are standard **`defclass`** slot definitions.

*class-option*s are standard **`defclass`** options. In addition the following options are recognized:

`(:interface` ***interface-name*****`)`**

This option is required. The component will implement the *interface-name*, which must be an Automation Collection interface, containing (at least) the standard properties **`Count`** and **`_NewEnum`**. The macro will define an implementation of these methods using information from the instance of the class to count and iterate.

`(:item-method` ***item-method-name*****`*)`**

When specified, a COM method named *item-method-name* will be defined that will look up items using the **`item-lookup-function`** from the instance.

If not specified, the method will be called **`Item`**. For Collections which do not have an item method, pass **`nil`** as the *item-method-name*.

Example

See also **`define-automation-component`**
`standard-automation-collection`

define-automation-component *Macro*

Summary Define an implementation class for a particular Automation component.

Package **`com`**

Signature **`define-automation-component`** ***class-name*** **(*superclass-name****)**
(*slot-specifier**)**
class-option*

Arguments

class-name A symbol naming the class to define.

superclass-name A symbol naming a superclass to inherit from.

slot-specifier A slot description as used by **`defclass`**.

class-option An option as used by **`defclass`**.

Description The macro **`define-automation-component`** defines a **`standard-class`** which is used to implement an Automation component. Normal **`defclass`** inheritance rules apply for slots and Lisp methods.

Each *superclass-name* argument specifies a direct superclass of the new class, which can be any **`standard-class`** provided that certain standard classes are included somewhere in the overall class precedence list. These standard classes depend on the other options and provide the default superclass list if none is specified. The following standard classes are available:

- **`standard-i-dispatch`** is always needed and provides a complete implementation of the **`i-dispatch`** interface, based on the type information in the type library.
- **`standard-i-connection-point-container`** is needed if there are any source interfaces specified (via the **`:coclass`** or **`:source-interfaces`** options). This provides a complete implementation of the Connection Point protocols.

*slot-specifier*s are standard **`defclass`** slot definitions.

*class-option*s are standard **`defclass`** options. In addition the following options are recognized:

`(:coclass` ***coclass-name*****`)`**

coclass-name is a symbol specifying the name of a coclass. If this option is specified then a class factory will be registered for this coclass, to create an instance of *class-name* when another application requires it. The component will implement the interfaces specified in the coclass definition and the default interface will be returned by the class factory.

Exactly one of **`:coclass`** and **`:interfaces`** must be specified.

`(:interfaces` ***interface-name*****`*)`**

Each *interface-name* specifies an Automation interface that the object will implement. The **`i-unknown`** and **`i-dispatch`** interfaces should not be specified because their implementations are automatically inherited from **`standard-i-dispatch`**. No class factory will be registered for *class-name*, so the only way to make instances is from with Lisp by calling **make-instance**.

Exactly one of **:coclass** and **:interfaces** must be specified.

(:source-interfaces *interface-name**)**

Each *interface-name* specifies a source interface on which the object allows connections to be made. If the **:coclass** option is also specified, then the interfaces flagged with the **source** attribute are used as the default for the **:source-interfaces** option.

When there are event interfaces, the component automatically implements the **IConnectionPointContainer** interface. The supporting interfaces **IEnumConnectionPoints**, **IConnectionPoint** and **IEnumConnections** are also provided automatically.

(:extra-interfaces *interface-name**)**

Each *interface-name* specifies a COM interface that the object will implement, in addition to the interfaces implied by the **:coclass** option. This allows the object to implement other interfaces not mentioned in the type library.

Use **define-com-method**, **define-dispinterface-method** or **com-object-dispinterface-invoke** to define methods in the interfaces implemented by the component. See also Section 1.8.4, "Unimplemented methods".

Example

```
(define-automation-component c-test-suite-1 ()
    ((prop3 :initform nil)
     (interface-4-called :initform nil))
  (:coclass test-suite-component)
  )
```

See also

define-com-method
define-dispinterface-method
com-object-dispinterface-invoke
standard-i-dispatch
standard-i-connection-point-container
define-automation-collection

define-dispinterface-method *Macro*

Summary

The **define-dispinterface-method** macro is used to define a *dispinterface* method.

Package

com

Signature

define-dispinterface-method ***method-spec*** **(*class-spec* . *lambda-list*)** ***form**** **=>** ***value***

method-spec **::=** ***method-name*** | **(*interface-name method-name*)**

class-spec **::=** **(*this class-name*)**

Arguments

method-spec Specifies the method to be defined.

method-name A symbol naming the method to define.

interface-name A symbol naming the interface of the method to define. This is only required if the implementation class *class-name* has more than one method with the given *method-name*.

class-spec Specifies the implementation class and variables bound to the object with in the *forms*.

this	A symbol which will be bound to the COM object whose method is being invoked.
class-name	A symbol naming the COM implementation class for which this method is defined.
lambda-list	A simple lambda list. That is, a list of parameter names.
form	Forms which implement the method. The value of the final form is returned as the result of the method.
value	The value to be returned to the caller.

Description

The macro **`define-dispinterface-method`** defines a *dispinterface* method that implements the method *method-name* for the Automation implementation class *class-name*. The extended *method-spec* syntax is required if *class-name* implements more than one interface with a method called *method-name* (analogous to the C++ syntax **`InterfaceName::MethodName`**).

The symbol *this* is bound to the instance of the Automation implementation class on which the method is being invoked.

The number of parameter in *lambda-list* must match the declaration in the type library. Each *in* and *in-out* parameter is bound to the value passed to **`IDispatch::Invoke`**, converted to the type specified by the method declaration and then converted to Lisp objects as specified in Section 3.3.3, “Data conversion when calling Automation methods”. For missing values the value of the parameter is **`:not-found`**. For *out* and *in-out* arguments, the corresponding parameter should be set by the forms to contain the value to be returned to the caller and will be converted to an automation value as specified in Section 3.3.3, “Data conversion when calling Automation methods”.

The *value* should be a value which can be converted to the appropriate return type as the primary value of the method and will be converted to an automation value as specified in Section 3.3.3, “Data conversion when calling Automation methods”. It is ignored for methods that are declared as returning void.

Note: the **`define-com-method`** macro should be used to implement methods in *dual* interfaces.

See also **`define-com-method`**
`com-object-dispinterface-invoke`

disconnect-standard-sink *Function*

Summary Releases a standard sink object, stopping the events.

Package **`com`**

Signature **`disconnect-standard-sink`** ***sink*** **`=>`** ***result***

Arguments *sink* A standard sink object.

Values *result* **`t`** or **`nil`**.

Description The function **`disconnect-standard-sink`** releases a standard sink object. This is one of the objects in the list returned by **`set-i-dispatch-event-handler`** which represents a connection it made.

`disconnect-standard-sink` stops the events that pass through *sink*.

result is **`t`** if the sink was released.

See also **`create-instance-with-events`**
`set-i-dispatch-event-handler`

do-collection-items *Macro*

Summary Iterates over the items of an Automation Collection.

Package `com`

Signature **`do-collection-items (`*item collection*`)` *form****

Arguments

item	A symbol bound to each item in the collection in turn.
collection	A form which is evaluated to yield a COM **`i-dispatch`** interface pointer that implements the collection protocol.
form	A form to be evaluated.

Description The **`do-collection-items`** macro executes each *form* in turn, with *item* bound to each item of the *collection*.

Note that for collections whose items are interface pointers, the *form*s must arrange for each pointer to be released when no longer needed. The *collection* should be a COM interface pointer for an **`i-dispatch`** interface that implements the Collection protocol. The items are converted to Lisp as specified in Section 3.3.3, “Data conversion when calling Automation methods”.

Example For example, to iterate over the **`Table`** objects from the **`Tables`** collection of a **`MyDocument`** interface pointer

```
(with-temp-interface (tables)
    (call-dispatch-get-property
       (doc my-document tables))
  (do-collection-items (table tables)
    (inspect-the-table table)
    (release table)))
```

See also **`call-dispatch-method`**

do-connections *Macro*

Summary Iterates over the sinks for a given Automation component object.

Package `com`

Signature

```
do-connections ((sink interface-name &key
                  dispatch automation-dispatch)
                 container)
                form*
```

Arguments

sink A symbol which will be bound to each sink interface pointer.

interface-name A symbol naming the sink interface.

dispatch A symbol which will be bound to a local macro that invokes a method from the sink interface as if by `with-com-interface`.

automation-dispatch

A symbol which will be bound to a local macro that invokes a method from the sink interface as if by `with-dispatch-interface`.

container An instance of a component class that has *interface-name* as one of its source interfaces.

form A form to be evaluated.

Description The macro `do-connections` provides a way to iterate over all the sink interface pointers for the source interface *interface-name* in the connection point container *container*. The *container* must be a subclass of `standard-i-connection-point-container`. Each *form* is evaluated in turn with *sink* bound to each interface pointer. If *dispatch* is given, it is defined as a local macro invoking the COM interface *interface-name* as if by `with-com-interface`. If *automation-dispatch* is given, it is defined as a local macro invoking the Automation interface *interface-name* as if by **with-dispatch-interface**.

Within the scope of **do-connections** you can call the local function **discard-connection** which discards the connection currently bound to *sink*. This is useful when an error is detected on that connection, for example when the client has terminated. The signature of this local function is

discard-connection &key *release*

release is a boolean defaulting to false. If *release* is true then **release** is called on *sink*.

Example

Suppose there is a source interface **i-clonable-events** with a method **on-cloned**. The following function can be used to invoke this method on all the sinks of an instance of a **clonable-component** class:

```
(defun fire-on-cloned (clonable-component)
  (do-connections ((sink i-clonable-events
                         :dispatch call-clonable)
                   clonable-component)
    (call-clonable on-cloned value)))
```

See also

with-dispatch-interface
with-com-interface
standard-i-connection-point-container

find-component-tlb *Function*

Summary

Returns the path of the type library associated with a component name.

Package

com

Signature

find-component-tlb *name* &key *version min-version max-version* => *path*

Arguments	*name*	A string.
	version	A string or **`nil`**.
	min-version	A string or **`nil`**.
	max-version	A string or **`nil`**.
Values	*path*	A string or **`nil`**.

Description

The function **`find-component-tlb`** returns the path of the type library associated with the component *name*.

name should be the name of a component (either a ProgID or a GUID).

If *version* is supplied, **`find-component-tlb`** finds only this version of the type library.

If *min-version* or *max-version*, or both of these, are supplied, they restrict which version of the type library can be found.

Each of *version*, *min-version* and *max-version*, if supplied, should be a string. The string should contain either one hexadecimal number or two hexadecimal numbers separated by a dot. The first number is the major version, the second is the minor version, which defaults to 0.

If *version* is not supplied, then **`find-component-tlb`** preferentially finds the the library version specified in the registry for the component (if any) if it fits the specification by *max-version* and / or *min-version*, otherwise it finds the earliest version in the range specified by *min-version* and *max-version*.

`find-component-tlb` returns **`nil`** if it fails to find the type library within the specified version constraints.

See also

`:midl-type-library-file`

find-component-value *Function*

Summary Searches the registry for values associated with a component.

Package `com`

Signature **`find-component-value`** ***name key-name*** **`=>`** ***result*, *root***

Arguments

name	A string.
key-name	A string or a keyword.

Values

result	A Lisp object.
root	A keyword.

Description

The function **`find-component-value`** searches the Windows registry for values associated with a component.

name should be the name of a component (either a ProgID or a GUID).

key-name should name a registry key. If it is a string, it should match the key name in the registry. Otherwise *key-name* can be one of the following keywords:

`:library` Returns the library that implements the component (if any)

`:inproc-server32`
As for **`:library`**

`:local-server32`
Returns the executable that implements the component (if any)

`:version` Returns the version

`:prog-id` Returns the ProgID

`:version-independent-prog-id`
Returns the version-independent ProgId

`:type-lib`	Returns the GUID of the type library

`find-component-value` returns the value *result* associated with the given *key-name* in the registry for component *name*. If a value is found., then there is a second returned value *root* which is either **`:local-machine`** or **`:user`**, indicating the branch of the registry in which the value was found.

`find-component-value` simply returns **`nil`** if it fails to find the information.

When running on 64-bit Windows, 32-bit LispWorks looks in the 32-bit registry view and 64-bit LispWorks looks in the 64-bit registry view. LispWorks does not change the registry reflection settings.

Examples

```
(com:find-component-value "shell.explorer" :version)
```

get-error-info *Function*

Summary

Retrieves the error information for the current Automation method.

Package

`com`

Signature

`get-error-info &key` ***errorp fields*** **`=>`** ***field-value****

Arguments

errorp	A boolean. If true and an error occurs while retrieving the error information, then an error of type **`com-error`** is signalled. Otherwise **`nil`** is returned.
fields	A list of keywords specifying the error information fields to return.

Values

*field-value**	Values corresponding to the *fields* argument.

Description The function `get-error-info` allows the various components of the error information to be retrieved for the last Automation method called. The *fields* should be a list of the following keywords, to specify which fields of the error information should be returned:

`:iid`	A `refguid` object.
`:source`	A string specifying the ProgID.
`:description`	A string describing the error.
`:help-file`	A string giving the help file's path.
`:help-context`	An integer giving the help context id.

A *field-value* will be returned for each *field* specified. The *field-value* will be `nil` if the *field* is does not have a value.

Example

```
(multiple-value-bind (source description)
    (get-error-info :fields '(:source :description))
  (error "Failed with '~A' in ~A" description source))
```

See also

`set-error-info`
`call-dispatch-method`
`com-error`

get-i-dispatch-name *Function*

Summary Returns the foreign name of an `i-dispatch` interface.

Package `com`

Signature `get-i-dispatch-name` ***i-dispatch*** `=>` ***name***

Arguments *i-dispatch* An `i-dispatch` interface.

Values *name* A string.

Description The function **`get-i-dispatch-name`** returns the foreign name of an **`i-dispatch`** interface. That is, it obtains the first return value of **`ITypeInfo::GetDocumentation`**.

Example To implement code like this:

```
If TypeOf objMap.Selection Is Pushpin Then
...
```

you would need something like:

```
(if (equalp (COM:get-i-dispatch-name selection)
            "PushPin")
    ...)
```

get-i-dispatch-source-names *Function*

Summary Returns the source names associated with an **`i-dispatch`** interface.

Package **`com`**

Signature **`get-i-dispatch-source-names`** ***i-dispatch*** **`&key`** ***all coclass*** **`=>`** ***source-names***

Arguments

i-dispatch	An **`i-dispatch`** interface.
all	A generalized boolean, default value false.
coclass	The coclass to use, or **`nil`**.

Values

source-names	A list.

Description The function **`get-i-dispatch-source-names`** returns the source names that are associated with the **`i-dispatch`** interface *i-dispatch*, which will be used by **`set-i-dispatch-event-handler`**.

coclass and *all* are as described for **`set-i-dispatch-event-handler`**.

Note: If you need to call **`set-i-dispatch-event-handler`** repeatedly, then it is most efficient to call **`get-i-dispatch-source-names`** once and pass the result *source-names* to **`set-i-dispatch-event-handler`**. This is because **`set-i-dispatch-event-handler`** itself calls **`get-i-dispatch-source-names`** if its *source-names* argument is **`nil`**.

See also **`set-i-dispatch-event-handler`**

i-dispatch *COM Interface Type*

Summary The Lisp name for the **`i-dispatch`** COM interface.

Package **`com`**

Description The symbol **`i-dispatch`** is the name given to the **`i-dispatch`** COM interface within Lisp. The name results from the standard mapping described in Section 1.3, "The mapping from COM names to Lisp symbols".

Examples

```
(query-interface ptr 'i-dispatch)
```

See also **`i-unknown`**
`standard-i-dispatch`

interface-connect *Function*

Summary Connects a sink interface pointer to the source of events in another COM interface pointer.

Package **`com`**

Signature **`interface-connect`** ***interface-ptr iid sink-ptr*** **`&key`** ***errorp*** **`=>`** ***cookie***

Arguments	*interface-ptr*	A COM interface pointer that provides the source interface *iid*.
	iid	The iid of the source interface to be connected. The iid can be a symbol naming the interface or a **`refguid`** foreign pointer.
	sink-ptr	A COM interface that will receive the events for the *iid*.
	errorp	A boolean. When false, errors connecting the *sink-ptr* will cause **`nil`** to be returned. Otherwise an error of type **`com-error`** will be signalled.
Values	*cookie*	An integer cookie associated with this connection.

Description Connects the COM interface *sink-ptr* to the connection point in *interface-ptr* that is named by *iid*.

Example Suppose there is an interface pointer **`clonable`** which provides a source interface **`i-clonable-events`**, then the following form can be used to connect an implementation of this source interface **`sink`**:

```
(setq cookie
      (interface-connect clonable
                         'i-clonable-events
                         sink))
```

See also

`interface-disconnect`
`refguid`
`com-error`

interface-disconnect *Function*

Summary Disconnect a sink interface pointer from the source of events in another COM interface pointer.

Package `com`

Signature **`interface-disconnect &key`** ***interface-ptr iid cookie*** **`&key`** ***errorp***
`=>` ***flag***

Arguments

interface-ptr A COM interface pointer that provides the source interface *iid*.

iid The iid of the source interface to be disconnected. The iid can be a symbol naming the interface or a **`refguid`** foreign pointer.

cookie The integer cookie associated with the connection to be disconnected.

errorp A boolean. When false, errors disconnecting the *cookie* will cause **`nil`** to be returned. Otherwise an error of type **`com-error`** will be signalled.

Values

flag A boolean, true for successful disconnection.

Description

Disconnects the connection for *cookie* from the connection point in *interface-ptr* that matches *iid*.

Example

Suppose there is an interface pointer **`clonable`** which provides a source interface **`i-clonable-events`**, then the following form can be used to disconnect an implementation of this source interface with cookie **`cookie`**:

```
(interface-disconnect clonable
                      'i-clonable-events
                      cookie)
```

See also

`interface-connect`
`refguid`
`com-error`

lisp-variant *Type*

Summary An object that contains a type and a value.

Package **com**

Accessors **lisp-variant-type**
lisp-variant-value

Description A **lisp-variant** is an object that contains a type and a value. The type and value are as described for the function **set-variant**.

See also **make-lisp-variant**
set-variant

invoke-dispatch-get-property *Function*

Summary Call a dispatch property getter method from an interface pointer.

Package **com**

Signature **invoke-dispatch-get-property** ***dispinterface-ptr name*** **&rest** ***args*** **=>** ***values***

Arguments

dispinterface-ptr	An Automation interface pointer.
name	A string or integer.
args	Arguments passed to the method.

Values

values	Values returned by the method.

Description The function **invoke-dispatch-get-property** is used to invoke an Automation property getter method from Lisp without needing to compile a type library as part of the application. This is similar to using

```
Dim var as Object
Print #output, var.Prop
```

in Microsoft Visual Basic and contrasts with the macro **call-dispatch-get-property** which requires a type library to be compiled.

The *dispinterface-ptr* should be a COM interface pointer for the **i-dispatch** interface. The appropriate Automation method, chosen using *name,* which is either a string naming the method or the integer id of the method. The *args* are converted to Automation values and are passed as the method's *in* and *in-out* parameters in the order in which they appear. The *values* returned consist of the primary value of the method (if not void) and the values of any *out* or *in-out* parameters. See Section 3.3.3, "Data conversion when calling Automation methods" for more details.

There is also **setf** expander for **invoke-dispatch-get-property**, which can be used as an alternative to the **call-dispatch-put-property** macro.

Example

For example, in order to get and set the **Width** property of an interface pointer in the variable **doc**:

```
(invoke-dispatch-get-property doc "Width")

(setf (invoke-dispatch-get-property
        doc "Width")
      10)
```

See also

invoke-dispatch-method
invoke-dispatch-put-property
call-dispatch-get-property

invoke-dispatch-method *Function*

Summary

Call a dispatch method from an interface pointer.

Package

com

Signature **`invoke-dispatch-method`** ***dispinterface-ptr name*** **`&rest`** ***args*** **`=>`** ***values***

Arguments

dispinterface-ptr	An Automation interface pointer.
name	A string or integer.
args	Arguments passed to the method.

Values

values	Values returned by the method.

Description The function **`invoke-dispatch-method`** is used to invoke an Automation method from Lisp without needing to compile a type library as part of the application. This is similar to using

```
Dim var as Object
var.Method(1,2)
```

in Microsoft Visual Basic and contrasts with the macro **`call-dispatch-method`** which requires a type library to be compiled.

The *dispinterface-ptr* should be a COM interface pointer for the **`i-dispatch`** interface. The appropriate Automation method, chosen using *name*, which is either a string naming the method or the integer id of the method. The *args* are converted to Automation values and are passed as the method's *in* and *in-out* parameters in the order in which they appear. The *values* returned consist of the primary value of the method (if not void) and the values of any *out* or *in-out* parameters. See Section 3.3.3, "Data conversion when calling Automation methods" for more details. If there is no Automation method with the given name, then a property getter with the same name is called if it exists, otherwise an error is signaled. The **`setf`** form of **`invoke-dispatch-method`** can be used to call property setter methods.

Example For example, in order to invoke the **`ReFormat`** method of an interface pointer in the variable **`doc`**:

```
(invoke-dispatch-method doc "ReFormat")
```

See also **invoke-dispatch-get-property**
invoke-dispatch-put-property
call-dispatch-method

invoke-dispatch-put-property *Function*

Summary Call a dispatch property setter method from an interface pointer.

Package **com**

Signature **invoke-dispatch-put-property** ***dispinterface-ptr name*** **&rest** ***args*** **=>** ***values***

Arguments

dispinterface-ptr	An Automation interface pointer.
name	A string or integer.
args	Arguments passed to the method.

Values

values	Values returned by the method.

Description The function **invoke-dispatch-put-property** is used to invoke an Automation property setter method from Lisp without needing to compile a type library as part of the application. This is similar to using

```
Dim var as Object
var.Prop = 2
```

in Microsoft Visual Basic and contrasts with the macro **call-dispatch-put-property** which requires a type library to be compiled.

The *dispinterface-ptr* should be a COM interface pointer for the **i-dispatch** interface. The appropriate Automation method, chosen using *name*, which is either a string naming the method or the integer id of the method. The *args* are converted to Automation values and are passed as the method's *in* and *in-out* parameters in the order in which they appear. The new value of the property should be the last argument. The *values* returned consist of the primary value of the method (if not void) and the values of any *out* or *in-out* parameters. See Section 3.3.3, “Data conversion when calling Automation methods” for more details.

Example

For example, in order to set the **Width** property of an interface pointer in the variable **doc**:

```
(invoke-dispatch-put-property doc "Width" 10)
```

See also

invoke-dispatch-method
invoke-dispatch-get-property
call-dispatch-put-property

make-lisp-variant *Function*

Summary

Returns a Lisp object that contains a type and a value.

Package

com

Signature

make-lisp-variant ***type*** **&optional** ***value*** **=>** ***lisp-variant***

Description

The function **make-lisp-variant** returns a **lisp-variant** object *lisp-variant* containing *type* and *value*.

lisp-variant can be passed as an argument to an Automation method, to give control over the VT code that the method sees. The meaning of *type* and *value* are as described for **set-variant**.

See also

lisp-variant
set-variant

:midl-type-library-file *Defsystem Member Type*

Summary — A defsystem member type that can be used to include a type library file in a Lisp system definition.

Package — **com**

Description — When a file is given the type **:midl-type-library-file**, compiling the system will compile the type library file to produce a fasl. Loading the system will load this fasl. The **:package** and **:mapping-options** keywords can specified as for **midl**.

The keyword **:component-name** *name-spec* can be supplied to specifiy that the source is the library specified by *name-spec*.

name-spec should be one of:

t	Means that the component name is the same as the module name.
A string	The name of the component.
A list	(*component-name keywords-and-values*) where the keywords and values are passed to **find-component-tlb** when looking for the actual library.

In all cases the module name, less anything after the last dot, is used as the default filename for the compiled file.

The keyword **:com** can be supplied to reduce the amount of code generated. For the details, see "Reducing the size of the converted library" on page 78.

Examples — To include the file **myfile.tlb** in a system, use

```
(defsystem my-system ()
  :members (("myfile.tlb"
             :type :midl-type-library-file)))
```

To compile the library associated with "OWC10.Spreadsheet", producing an object file in **OWC10.ofasl** put a clause like this in the defsystem form:

```
("OWC10.SPREADSHEET" :type :midl-type-library-file
                     :com :not-binary
                     :component-name t)
```

To compile the same library, but to a different object file, use:

```
("my-owc" :type :midl-type-library-file
          :com :not-binary
          :component-name "OWC10.SPREADSHEET")
```

To compile the same library, but using only version newer than 1.1, use a claus like this:

```
("my-owc" :type :midl-type-library-file
           :com :not-binary
           :component-name ("OWC10.SPREADSHEET"
                            :min-version "1.1"))
```

See also **find-component-tlb**

:midl-file

query-simple-i-dispatch-interface *Function*

Summary Queries the interface pointer from a **simple-i-dispatch** object using the type information from another interface.

Package **com**

Signature **query-simple-i-dispatch-interface** ***this*** **&key** ***related-dispatch*** **=>** ***interface-ptr*, *refguid***

Arguments

this	A **simple-i-dispatch** object.
related-dispatch	An **i-dispatch** interface pointer.

Values

interface-ptr	An interface pointer.
refguid	A **refguid**.

Description

The function **query-simple-i-dispatch-interface** is used to obtain an interface pointer from a **simple-i-dispatch** interface. The **simple-i-dispatch** contains the interface name provided using its **:interface-name** initarg, but it doesn't have the details of this interface, so **query-simple-i-dispatch-interface** must be able to find the details.

In the current implementation, the only way for the details to be found is by passing the *related-dispatch* argument. This should be an interface pointer from which type information about the interface name can be obtained.

The **query-simple-i-dispatch-interface** function returns two values, *interface-ptr* which is an interface pointer for the interface-name contained in *this* and *refguid*, which is the **refguid** of that interface-name.

A typical use of **query-simple-i-dispatch-interface** is to implement a sink interface for events from some other component. The interface pointer for that component is passed as the *related-dispatch* because that connects to the type library containing both interface definitions.

Before using **query-simple-i-dispatch-interface** directly, consider the functions **set-i-dispatch-event-handler** and **create-instance-with-events**, which provide an succinct way to provide an event callback.

See also

simple-i-dispatch
create-instance-with-events
set-i-dispatch-event-handler

set-error-info *Function*

Summary

Sets the error information for the current Automation method.

Package

com

Signature **`set-error-info &key`** ***iid source description help-file help-context***
`=>` ***error-code***

Arguments

iid — The iid of the interface that defined the error, or **`nil`** if none. The iid can be a symbol naming the interface or a **`refguid`** foreign pointer.

source — A string giving the ProgID for the class that raised the error, or **`nil`** if none.

description — A string giving the textual description of the error, or **`nil`** if none.

help-file — A string giving the path of the help file that describes the error, or **`nil`** if none.

help-context — An integer giving the help context id for the error, or **`nil`** if none.

Values

error-code — The error code **`DISP_E_EXCEPTION`** or nil if the error info could not be set.

Description

The function **`set-error-info`** allows the various components of the error information to be set for the current Automation method. It should only be called within the dynamic scope of the body of a **`define-com-method`** definition. The value **`DISP_E_EXCEPTION`** can be returned as the **`hresult`** of the method to indicate failure.

Examples

```
(define-com-method (i-robot rotate)
    ((this i-robot-impl)
     (axis :in)
     (angle-delta :in))
  (let ((joint (find-joint axis)))
    (if joint
        (progn
          (rotate-joint joint)
          S_OK)
      (set-error-info :iid 'i-robot
                      :description "Bad joint."))))
```

See also `define-com-method`
`get-error-info`
`refguid`
`hresult`

set-i-dispatch-event-handler *Function*

Summary Sets an event handler for an `i-dispatch` interface.

Package `com`

Signature
```
set-i-dispatch-event-handler
  (interface event-handler &key all coclass
                                event-object source-names)
                                                       => sinks
```

Arguments

interface	An `i-dispatch` interface.
event-handler	A function of four arguments.
all	A generalized boolean, default value false.
coclass	The coclass to use, or `nil`.
event-object	A Lisp object.
source-names	A list of "source" interface names, or `nil`.

Values

sinks	A list of objects representing the connections made.

Description The function `set-i-dispatch-event-handler` sets an event handler for the `i-dispatch` interface *interface*.

event-handler is a function of four arguments:

`event-handler` ***event-obj method-name method-type args***

event-obj is the value of *event-object* if this is non-nil. If *event-object* is **`nil`**, *event-obj* is the value of *interface*.

method-name is the method-name that has been called, which is the same as the "event" name in Visual Basic terminology.

method-type is the type of the method. For a normal "event" it is **`:method`**. *method-type* can also be **`:put`** or **`:get`** if the underlying "source" interface has "propput" or "propget" methods or properties.

args is an array containing the arguments to the method ("event"). This varies according to the method. For *out* or *in-out* arguments, it is possible to return a value by setting the corresponding value in the array.

The *all, coclass and source-names* arguments to **`set-i-dispatch-event-handler`** tell it which "source" interface or interfaces to use. In most cases, the default is correct.

If *all* is false, then only the "default" "source" is used. If *all* is true, then **`set-i-dispatch-event-handler`** uses all the source interfaces that the coclass defines.

coclass tells **`set-i-dispatch-event-handler`** which coclass to use, which is the same as the object in Visual Basic terminology.

If *coclass* is **`nil`**, it uses the first coclass in the type library that has the type of *interface* as a default interface, or if there is no such coclass, the first coclass that has this interface. In most of the cases this is the desired coclass.

If *coclass* is non-nil, it specifies which coclass to use. It can be a ProgID (for example **`"Word.Application"`**) or a coclass name or a coclass GUID. If the **`i-dispatch`** *interface* was created with **`create-instance`**, then the argument to **`create-instance`** is the correct coclass to use.

If *source-names* is non-nil, then it is a list of "source" interface names to use, and *all* and *coclass* are ignored. If source-names is nil, then **`set-i-dispatch-event-handler`** calls **`get-i-dis-`**

patch-source-names to calculate the "source" interface names.

sinks is a list of objects representing the connections that **set-i-dispatch-event-handler** made. When the events are no longer needed, they can be released by **disconnect-standard-sink**.

Note: **set-i-dispatch-event-handler** can be called more than once on the same **i-dispatch**, and this generates new connections each time. Therefore, if it is called more than once such that it uses the same source names, events will arrive more than once.

Note: if you need to call **set-i-dispatch-event-handler** repeatedly, then it is most efficient to call **get-i-dispatch-source-names** once and pass the result *source-names* to **set-i-dispatch-event-handler**.

Note: there is a useful function **create-instance-with-events** which combines **create-instance** and **set-i-dispatch-event-handler**.

See also **disconnect-standard-sink**
create-instance-with-events
get-i-dispatch-source-names

set-variant *Function*

Summary Sets the fields in a **VARIANT** pointer.

Package **com**

Signature **set-variant** ***variant type*** **&optional** ***value***

Arguments *variant* A foreign pointer to an object of type **VARIANT**.

type A keyword specifying the type of value.

value The value to store in *variant*.

Description

The function **set-variant** can be used to set the type and value of a **VARIANT** object. It is useful if the default type provided by the automatic conversion for **VARIANT** return values is incorrect. The value of meaning of *type* is an specified below

Value of *type*	**VT** code used	Expected type of *value*
nil	dynamic	any suitable
:empty	**VT_EMPTY**	ignored
:null	**VT_NULL**	ignored
:short	**VT_I2**	**integer**
:long	**VT_I4**	**integer**
:float	**VT_R4**	**single-float**
:double	**VT_R8**	**double-float**
:cy	**VT_CY**	
:date	**VT_DATE**	
:bstr	**VT_BSTR**	**string**
:dispatch	**VT_DISPATCH**	FLI pointer
:error	**VT_ERROR**	ignored
:bool	**VT_BOOL**	**nil** or non **nil**
:variant	**VT_VARIANT**	FLI pointer
:unknown	**VT_UNKNOWN**	FLI pointer
:decimal	**VT_DECIMAL**	
(:unsigned :char)	**VT_UI1**	**integer**
(:array . *type***)**	**VT_BYREF +** **VT** code for *type*	**array**
:array or **(:array** *array***)** or **(:array .** *types***)**	**VT_ARRAY +** **VT_VARIANT**	**array**
(:pointer *type2***)**	**VT_BYREF +** **VT** code for *type2*	FLI pointer

If *type* is `nil` then the actual VT code is chosen dynamically according to the Lisp type of *value* (see Table 3.1, page 81).

If *type* is a cons of the form `(:array .` *type*`)` for some keyword *type*, then *variant* is set to contain an array of objects of *type*. Each element of *value* is expected to be suitable for conversion to *type*.

If *type* is `:array` or another list starting with `:array` then *variant* is set to contain an array of `VARIANT` objects with the same dimensions as *value*. Each element of *value* is converted as if by calling `set-variant` with a type chosen as follows:

- If *type* is the symbol `:array`, then `nil` is passed as the element type.
- If *type* is of the form `(:array` *array*`)` then *array* should be an array with the same dimensions as *value*. The element type is taken from the corresponding element of *array*.
- If *type* is of the form `(:array .` *types*`)` then *types* should be a suitable value for the `:initial-contents` argument to `make-array` to make an array of types with the same dimensions as *value*. The element type is taken from the corresponding element of that array. In particular, if *value* is a `vector` of length *n* then *type* should be a list of the form `(:array` $type_1$ $type_2$ ... $type_n$`)`.

Examples

```
(set-variant v :null)

(set-variant v :short 10)

(set-variant v '(:pointer :short) ptr)

(set-variant v '(:array :short :int) #(1 2))
```

See also

`define-com-method`

simple-i-dispatch *Class*

Summary
A complete dynamic implementation of the **i-dispatch** interface.

Package
com

Superclasses
standard-i-dispatch

Subclasses
None

Initargs
:interface-name

The name of the interface to implement. See **query-simple-i-dispatch-interface** for details on how this is used.

:invoke-callback

A function that is called with four arguments whenever one of the interface's methods is invoked. The arguments are the callback object, the method name as a string, the method type (a keyword **:method**, **:get** or **:put**) and a vector of the method's arguments. The value returned by the function will be returned to the caller of the method See **com-object-dispinterface-invoke** for more details of the method name, type and arguments.

Accessors
simple-i-dispatch-invoke-callback

Readers
simple-i-dispatch-interface-name
simple-i-dispatch-refguid

Description
The class **simple-i-dispatch** provides a complete implementation of the **i-dispatch** interface, without requiring a type library to be parsed. The type information is obtained at run-time when **query-simple-i-dispatch-interface** is

called. The class inherits from **`standard-i-dispatch`** to provide the **`i-unknown`** interface.

The **`simple-i-dispatch-refguid`** reader can be used to return the **`refguid`** of the interface. This can only be called after **`query-simple-i-dispatch-interface`** has been called.

The implementation obtains the callback object argument to the *invoke-callback* by calling **`simple-i-dispatch-callback-object`** with the **`simple-i-dispatch`** object. The default method returns the **`simple-i-dispatch`** object itself, but this method can be overridden for subclasses to return some other object.

Before using **`simple-i-dispatch`** directly, consider the functions **`set-i-dispatch-event-handler`** and **`create-instance-with-events`**, which provide an succinct way to provide an event callback.

See also

`query-simple-i-dispatch-interface`
`simple-i-dispatch-callback-object`
`standard-i-dispatch`
`i-dispatch`
`capi:ole-control-pane-simple-sink`

simple-i-dispatch-callback-object *Generic Function*

Summary

A generic function that can be implemented to modify the first argument to the *invoke-callback* in **`simple-i-dispatch`**.

Package

`com`

Signature

this **`=>`** ***object***

Method Signature

`(`*this*` simple-i-dispatch) => `*this*

Arguments

this An object of type **`simple-i-dispatch`**.

Values *object* The callback object to be pass as the first argument to the *invoke-callback* of *this*.

Description The generic function **simple-i-dispatch-callback-object** is called by the implementation of **simple-i-dispatch** to obtain the callback object (first argument) to its *invoke-callback*. This allows the object to be computed in some way by subclassing **simple-i-dispatch** and implementing a method on **simple-i-dispatch-callback-object** specialized for the subclass.

The pre-defined primary method specializing on **simple-i-dispatch** always returns its argument.

Example When the function **my-dispatch-callback** below is called, its first argument will be the *useful-object* passed to **make-my-dispatch**.

```
(defclass my-dispatch (simple-i-dispatch)
  ((useful-object :initarg :useful-object)))

(defmethod simple-i-dispatch-callback-object
    ((this my-dispatch))
  (slot-value this 'useful-object))

(defun make-my-dispatch (useful-object)
  (make-instance
   'my-dispatch
   :useful-object useful-object
   :invoke-callback 'my-dispatch-callback
   :interface-name "MyDispatchInterface"))
```

See also **simple-i-dispatch**

standard-automation-collection *Class*

Summary A framework for implementing Automation collections.

Package **com**

Superclasses **standard-i-dispatch**

Initargs **:count-function**

A function of no arguments that should return the number of items in the collection. This initarg is required.

:items-function

A function of no arguments that should return a sequence of items in the collection. This function is called by the implementation of **_NewEnum** and the sequence is copied. Exactly one of **:items-function** and **:item-generator-function** must be specified.

:item-generator-function

A function of no arguments that should return an *item generator*, which will generate the items in the collection. See below for more details. Exactly one of **:items-function** and **:item-generator-function** must be specified.

:data-function

A function called on each item that the **:items-function** or **:item-generator-function** returns. This is called when iterating, to produce the value that is returned to the caller.

:item-lookup-function

A function which takes a single argument, an integer or a string specifying an item. The function should return the item specified. This initarg is required if the :**item-method** option is non-nil in **define-automation-collection**.

Description

The class `standard-automation-collection` provides a framework for implementing Automation collections. These typically provide a `Count` property giving the number of objects in the collect, a `_NewEnum` property for iterating over the element of the collection method and optionally an `Item` method for finding items by index or name.

The `:count-function` initarg specifies a function to count the items of the collection and is invoked by the implementation of the `Count` method.

Exactly one of the initargs `:item-function` and `:item-generator-function` must be specified to provide items for the implementation of the `IEnumVARIANT` instance returned by the `_NewEnum` method.

If `:items-function` is specified, then it will be called once when `_NewEnum` is called and should return a sequence of the items in the collection. This sequence is copied, so can be modified by the program without affecting the collection.

If `:item-generator-function` is specified, it should be an *item generator* that will generate all the items in the collection. It will be called once with the argument `:clone` when `_NewEnum` is called and then by the implementation of the resulting `IEnumVARIANT` interface. An *item generator* is a function of one argument which specifies what to do:

`:next`	Return two values: the next item and `t`. If there are no more items, return `nil` and `nil`.
`:skip`	If there are no more items, return `nil`. Otherwise skip the current item and return `t`.
`:reset`	Reset the generator so the first item will be returned again.
`:clone`	Return a copy of the *item generator*. The copy should have the same current item.

The `:data-function` initarg should be function to convert each item returned by the `:items-function` or the item generator into a value whose type is compatible with Automation (see Table 3.1, page 81). The default function is **identity**.

Example — See the example in the directory

examples/com/automation/collections/

See also — **define-automation-collection**
standard-i-dispatch
i-dispatch

standard-i-connection-point-container *Class*

Summary — A complete implementation of the Connection Point protocol.

Package — **com**

Superclasses — **standard-i-unknown**

Description — The class **standard-i-connection-point-container** provides a complete implementation of the Connection Point protocols. It implements the **IConnectionPointContainer** interface and creates connection points for each interface given by the **:outgoing-interfaces** initarg.

If a class defined with **define-automation-component** macro specifies the **:source-interfaces** option or has interfaces with the "source" attribute in its coclass then it must inherit from **standard-i-connection-point-container** somehow. **define-automation-component** passes the appropriate initargs to initialize the class.

The macro **do-connections** can be used to iterate over the connections (sinks) for a given interface.

Example — Given the class definition

```
(define-automation-component clonable-component ()
    ()
  (:interfaces i-clonable)
  (:source-interfaces i-clonable-events)
  )
```

then

```
(typep (make-instance 'clonable-component)
       'standard-i-connection-point-container)
=> t
```

See also

define-automation-component
standard-i-dispatch
do-connections
define-automation-collection
standard-i-unknown
i-dispatch

standard-i-dispatch *Class*

Summary

A complete implementation of the **i-dispatch** interface.

Package

com

Superclasses

standard-i-unknown

Subclasses

standard-automation-collection
simple-i-dispatch

Description

The class **standard-i-dispatch** provides a complete implementation of the **i-dispatch** interface, based on the type information in the type library. In addition, the **i-support-error-info** interface is implemented to support error information. **standard-i-dispatch** inherits from **standard-i-unknown** to provide the **i-unknown** interface.

All classes defined with the **define-automation-component** and **define-automation-collection** macros must inherit from **standard-i-dispatch** somehow. These macros pass the appropriate initargs to initialize the class.

Example

Given the class definition

```
(define-automation-component document-impl ()
    ()
  (:coclass document)
  )
```

then

```
(typep (make-instance 'document-impl)
       'standard-i-dispatch)
=> t
```

See also

define-automation-component
define-automation-collection
standard-i-connection-point-container
standard-i-unknown
i-dispatch

with-coclass *Macro*

Summary

Executes a body of code with a temporary instance of a coclass.

Package

com

Signature

with-coclass ***disp form**** **=>** ***values***

disp **::=** **(*dispatch-function coclass-name*
&key *interface-name punk clsctx*)**

Arguments

disp The names of the dispatch function, coclass etc.

dispatch-function	A symbol which will be defined as a macro, as if by **`with-dispatch-interface`**. The macro can be used by the *forms* to invoke the Automation methods of the component.
coclass-name	A symbol which names the coclass. It is not evaluated.
interface-name	A symbol naming an interface in the coclass. It is not evaluated.
punk	A symbol which will be bound to the interface pointer.
clsctx	A CLSCTX value, which defaults to **`CLSCTX_SERVER`**.
form	A form to be evaluated.

Values

values — The values returned by the last *form*.

Description

Calls **`create-object`** to make an instance of the coclass named by the symbol *coclass-name*. If *interface-name* is given then that interface is queried from the component, otherwise the default interface is queried. Each *form* is evaluated in turn with *dispatch-function* bound of a local macro for invoking methods on the interface, as if by **`with-dispatch-interface`**. After the forms have been evaluated, the interface pointer is released. If *punk* is given, it will be bound to the interface pointer while the forms are being evaluated.

Example

If a type library containing the coclass **`TestComponent`** has been converted to Lisp, then following can be used to make an instance of component and invoke the **`Greet()`** method on the default interface.

```
(with-coclass (call-it test-component)
  (call-it greet "hello"))
```

See also

`create-object`

with-dispatch-interface *Macro*

Summary Used to simplify invocation of several methods from a particular Automation interface pointer.

Package `com`

Signature **`with-dispatch-interface`** ***disp dispinterface-ptr form**** **`=>`** ***values***

disp **`::=`** **`(`*****dispatch-function dispinterface-name*****`)`**

Arguments

disp The names of the dispatch function and Automation interface.

dispatch-function A symbol which will be defined as a macro, as if by **`macrolet`**. The macro can be used by the *form*s to invoke the methods on *dispinterface-ptr*.

dispinterface-name
A symbol which names the Automation interface. It is not evaluated.

dispinterface-ptr A form which is evaluated to yield a COM **`i-dispatch`** interface pointer.

form A form to be evaluated.

Values *values* The values returned by the last *form*.

Description When the macro **`with-dispatch-interface`** evaluates the *form*s, the local macro *dispatch-function* can be used to invoked the methods for the Automation interface *dispinterface-name*, which should be the type or a supertype of the actual type of the Automation interface pointer *dispinterface-ptr*.

The *dispatch-function* macro has the following signature:

dispatch-function method-name arg* **`=>`** ***values***

where

method-name	A symbol which names the method. It is not evaluated.
arg	Arguments to the method (see Section 3.3.3, "Data conversion when calling Automation methods" for details).
values	Values from the method (see Section 3.3.3, "Data conversion when calling Automation methods" for details).

Example

For example, in order to invoke the **ReFormat** method of a **MyDocument** interface pointer

```
(with-dispatch-interface (call-doc my-document) doc
  (call-doc re-format))
```

See also

call-dispatch-method

5

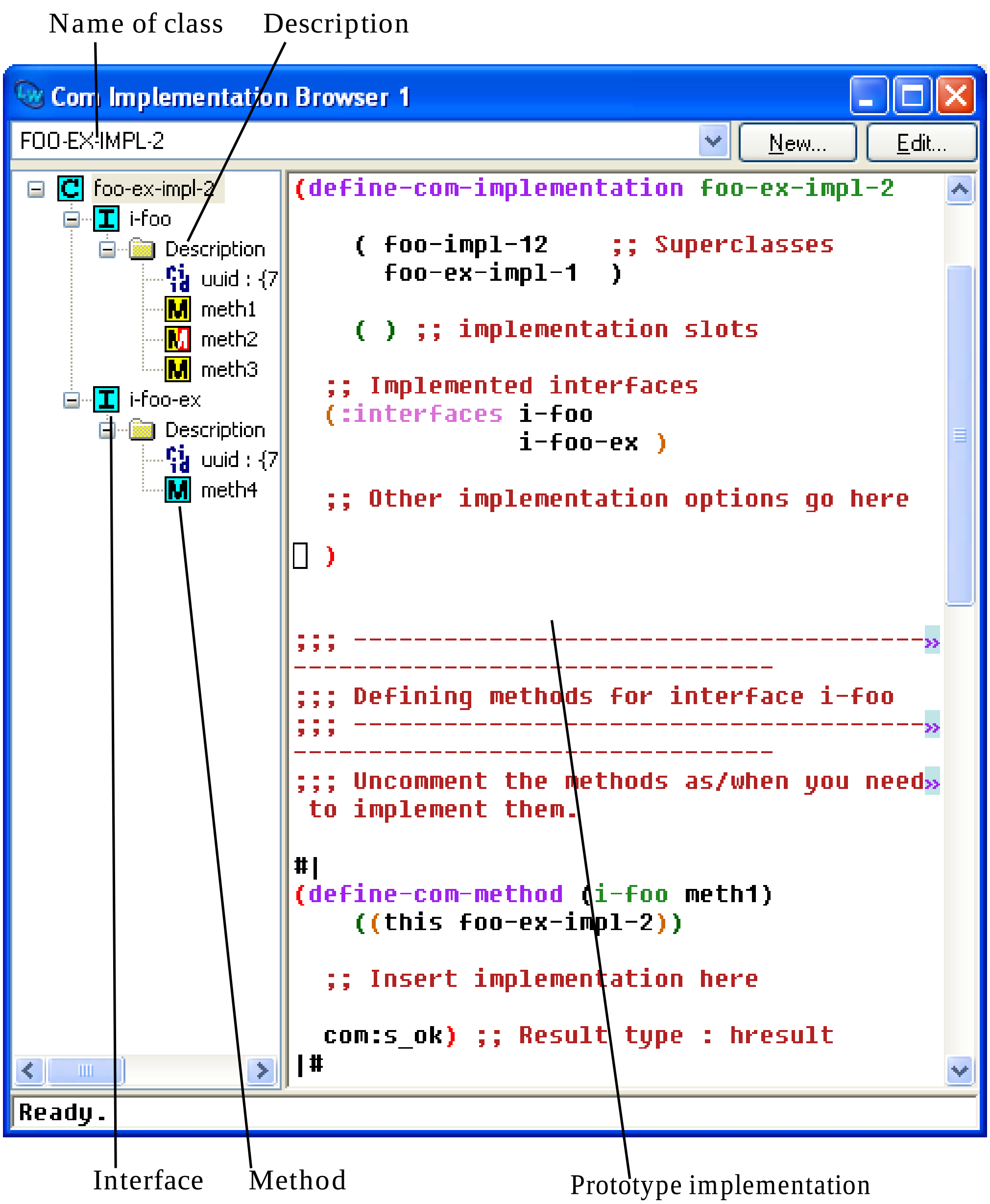
Tools

The tools described in this chapter extend the LispWorks IDE to help with debugging applications using COM/Automation. See the *LispWorks IDE User Guide* for more details of common operations that can be performed within these tools. The sections below describe each tool.

5.1 The COM Implementation Browser

The COM Implementation Browser allows prototype code for COM implementation classes to be viewed and created. This is useful when writing COM methods because it provides a template for the method names and arguments.

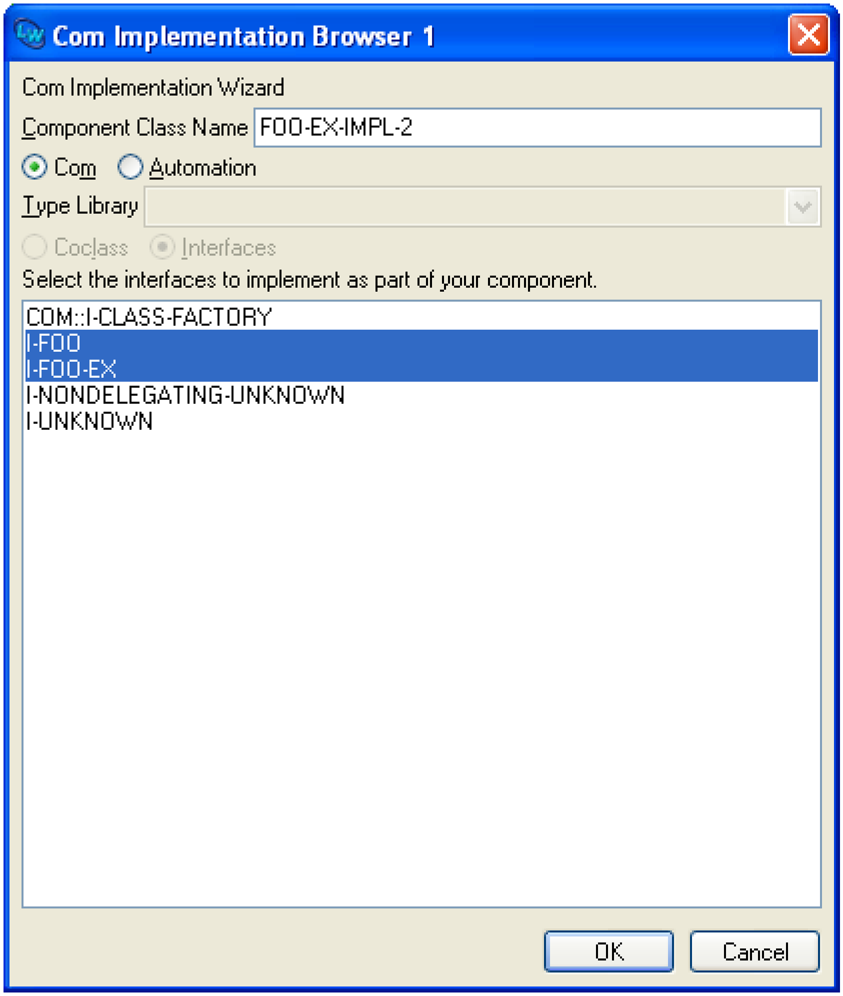
To start the tool, choose **Tools > Com Implementation Browser** from the LispWorks podium.



At the top of the window is a drop down list a class names. Choosing an item from this list will set the contents of the Description panel to show that class at the root of the tree, with subitems for each COM interface that it implements. The COM interfaces have subitems for their uuids and methods. The icon used for a method in the tree indicates the status of its implementation: red means not implemented (see Section 1.8.4 on page 17), yellow means inherited from a superclass (see Section 1.8.5 on page 17), red and yellow means an inherited unimplemented method and cyan means a method implemented directly in the named class.

Selecting an item in the Description pane will display a prototype implementation for that part of the class, using the appropriate macros for COM and Automation classes.

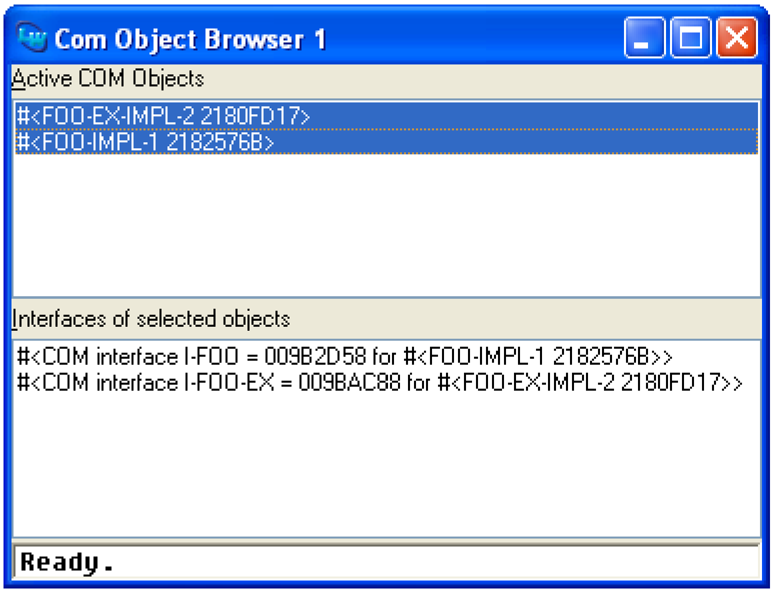
The **New** and **Edit** buttons allow prototype classes to be constructed and modified. Such classes are shown in the list of class names as **Example class...** and are not actually defined, but the prototype code can be copied into a file and evaluated to provide a starting point for an implementation. Clicking **New** or **Edit** displays a dialog as shown below.



The class name is displayed at the top and can be edited. For COM object classes, the list at the bottom of the dialog shows the COM interfaces that the class will implement. For Automation interfaces, a type library must be chosen from the drop-down list and one of the **Coclass** or **Interfaces** options selected to show the list of coclasses or interfaces that the class will implement. Click **OK** to confirm your choice or **Cancel** to discard it.

5.2 The COM Object Browser

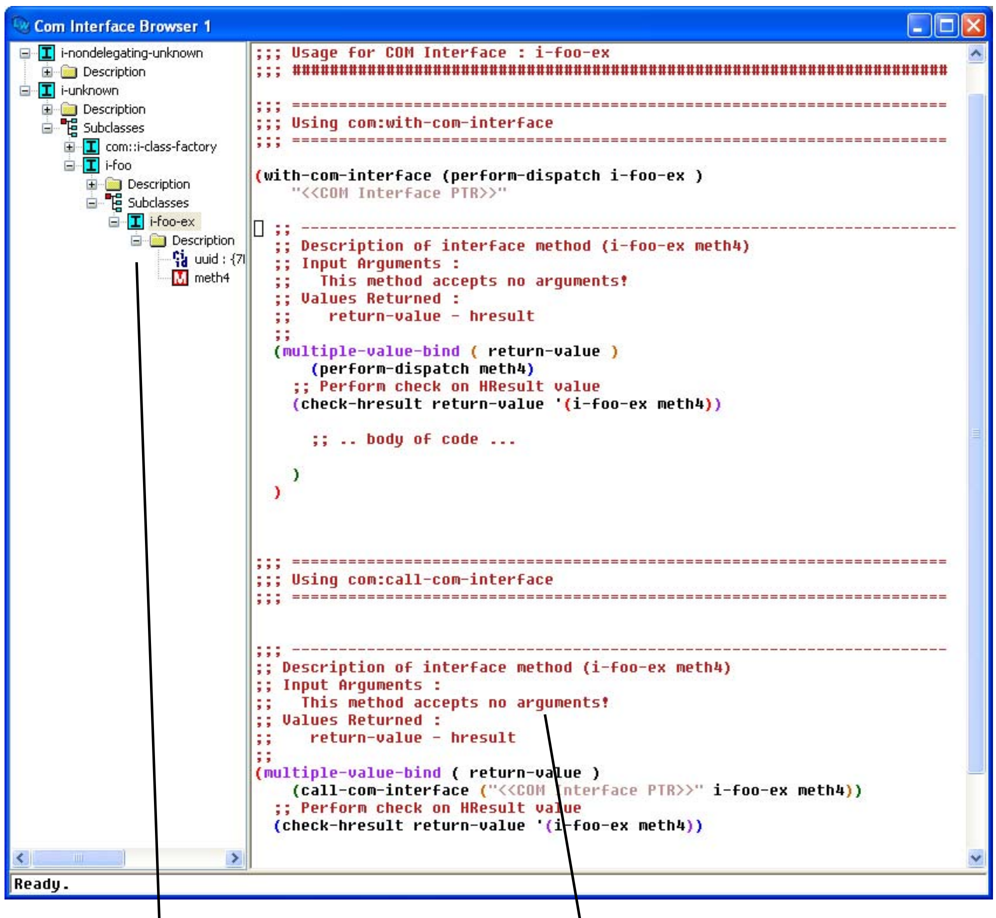
The COM Object Browser is used view COM objects for the classes implemented by Lisp. To start the tool, choose **Tools > Com Object Browser** from the LispWorks podium.



The **Active COM Objects** list shows all the Lisp objects that are known to the COM runtime system. Selecting objects from this list will list the COM interface pointers that have been queried for these objects. Double clicking on either list will inspect the data. Use the **Works > Object** menu or the context menu to perform other operations on the selected COM Objects.

5.3 The COM Interface Browser

The COM Interface Browser allows the interfaces that have been converted to FLI definitions to be viewed. To start the tool, choose **Tools > Com Interface Browser** from the LispWorks podium.



Interfaces and methods   Prototype code for invoking methods

The left hand pane shows a tree of the interfaces, with subitems for their uuids and methods. Selecting an item will cause the right-hand pane to show prototype code for invoking the method(s) selected.

5.4 Editor extensions

The LispWorks editor has been enhanced to support COM.

5.4.1 Inserting GUIDs

The editor command **`Insert GUID`** can be used to insert a new GUID at the current point. The GUID is made by calling **`CoCreateGUID`**.

5.4.2 Argument lists

The editor command **`Function Arglist`** (**`Alt+=`**) has been extended to show the arguments for all COM methods which match the function name.

Index