Copyright and Trademarks


Version 8.0

December 2021

Copyright © 2021 by LispWorks Ltd.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of LispWorks Ltd.

The information in this publication is provided for information only, is subject to change without notice, and should not be construed as a commitment by LispWorks Ltd. LispWorks Ltd assumes no responsibility or liability for any errors or inaccuracies that may appear in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

LispWorks and KnowledgeWorks are registered trademarks of LispWorks Ltd.

Adobe and PostScript are registered trademarks of Adobe Systems Incorporated. Other brand or product names are the registered trademarks or trademarks of their respective holders.


The XP Pretty Printer bears the following copyright notice, which applies to the parts of LispWorks derived therefrom: Copyright © 1989 by the Massachusetts Institute of Technology, Cambridge, Massachusetts.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. M.I.T. disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall M.I.T. be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

LispWorks contains part of ICU software obtained from http://source.icu-project.org and which bears the following copyright and permission notice:

ICU License - ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE

Copyright © 1995-2006 International Business Machines Corporation and others. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Copyright and Trademarks

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder. All trademarks and registered trademarks mentioned herein are the property of their respective owners.

US Government Restricted Rights
The LispWorks Software is a commercial computer software program developed at private expense and is provided with restricted rights. The LispWorks Software may not be used, reproduced, or disclosed by the Government except as set forth in the accompanying End User License Agreement and as provided in DFARS 227.7202-1(a), 227.7202-3(a) (1995), FAR 12.212(a)(1995), FAR 52.227-19, and/or FAR 52.227-14 Alt III, as applicable. Rights reserved under the copyright laws of the United States.

<table>
<thead>
<tr>
<th>Address</th>
<th>Telephone</th>
<th>Fax</th>
</tr>
</thead>
<tbody>
<tr>
<td>LispWorks Ltd</td>
<td>From North America:</td>
<td></td>
</tr>
<tr>
<td>St. John's Innovation Centre</td>
<td>877 759 8839 (toll-free)</td>
<td></td>
</tr>
<tr>
<td>Cowley Road</td>
<td>From elsewhere:</td>
<td></td>
</tr>
<tr>
<td>Cambridge</td>
<td>+44 1223 421860</td>
<td></td>
</tr>
<tr>
<td>CB4 0WS</td>
<td>From North America:</td>
<td></td>
</tr>
<tr>
<td>England</td>
<td>617 812 8283</td>
<td></td>
</tr>
<tr>
<td></td>
<td>From elsewhere:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+44 870 2206189</td>
<td></td>
</tr>
</tbody>
</table>

www.lispworks.com
Contents

Preface  

1 Using COM  

2 COM Reference Entries  

add-ref  
automation-server-command-line-action  
automation-server-main  
automation-server-top-loop  
call-com-interface  
call-com-object  
check-hresult  
co-create-guid  
co-initialize  
com-error  
com-interface  
com-interface-refguid  
com-object  
com-object-destructor  
com-object-from-pointer  
com-object-initialize  
com-object-query-interface  
co-task-mem-alloc  
co-task-mem-free  
co-uninitialize  
create-instance  
define-com-implementation  

3 Using Automation   80

3.1 Including Automation in a Lisp application  80
3.2 Starting a remote Automation server  81
3.3 Calling Automation methods  81
3.4 Implementing Automation interfaces in Lisp  84
3.5 Examples of using Automation  86
## 4 Automation Reference Entries  87

<table>
<thead>
<tr>
<th>Entry</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>call-dispatch-get-property</td>
<td>87</td>
</tr>
<tr>
<td>call-dispatch-method</td>
<td>88</td>
</tr>
<tr>
<td>call-dispatch-put-property</td>
<td>89</td>
</tr>
<tr>
<td>com-dispatch-invocation-exception-error</td>
<td>90</td>
</tr>
<tr>
<td>com-dispatch-invocation-exception-error-info</td>
<td>91</td>
</tr>
<tr>
<td>com-object-dispinterface-invocation</td>
<td>92</td>
</tr>
<tr>
<td>create-instance-with-events</td>
<td>93</td>
</tr>
<tr>
<td>create-object</td>
<td>94</td>
</tr>
<tr>
<td>define-automation-collection</td>
<td>95</td>
</tr>
<tr>
<td>define-automation-component</td>
<td>96</td>
</tr>
<tr>
<td>define-dispinterface-method</td>
<td>98</td>
</tr>
<tr>
<td>disconnect-standard-sink</td>
<td>100</td>
</tr>
<tr>
<td>do-collection-items</td>
<td>100</td>
</tr>
<tr>
<td>do-connections</td>
<td>101</td>
</tr>
<tr>
<td>find-component-tlb</td>
<td>102</td>
</tr>
<tr>
<td>find-component-value</td>
<td>103</td>
</tr>
<tr>
<td>get-active-object</td>
<td>105</td>
</tr>
<tr>
<td>get-error-info</td>
<td>106</td>
</tr>
<tr>
<td>get-i-dispatch-name</td>
<td>107</td>
</tr>
<tr>
<td>get-i-dispatch-source-names</td>
<td>108</td>
</tr>
<tr>
<td>i-dispatch</td>
<td>108</td>
</tr>
<tr>
<td>interface-connect</td>
<td>109</td>
</tr>
<tr>
<td>interface-disconnect</td>
<td>110</td>
</tr>
<tr>
<td>invoke-dispatch-get-property</td>
<td>111</td>
</tr>
<tr>
<td>invoke-dispatch-method</td>
<td>112</td>
</tr>
<tr>
<td>invoke-dispatch-put-property</td>
<td>113</td>
</tr>
<tr>
<td>lisp-variant</td>
<td>114</td>
</tr>
<tr>
<td>make-lisp-variant</td>
<td>115</td>
</tr>
<tr>
<td>:midl-type-library-file</td>
<td>116</td>
</tr>
<tr>
<td>print-i-dispatch-method</td>
<td>117</td>
</tr>
<tr>
<td>query-simple-i-dispatch-interface</td>
<td>118</td>
</tr>
<tr>
<td>register-active-object</td>
<td>119</td>
</tr>
<tr>
<td>revoke-active-object</td>
<td>120</td>
</tr>
<tr>
<td>set-error-info</td>
<td>120</td>
</tr>
<tr>
<td>set-i-dispatch-event-handler</td>
<td>121</td>
</tr>
<tr>
<td>set-variant</td>
<td>123</td>
</tr>
<tr>
<td>simple-i-dispatch</td>
<td>125</td>
</tr>
<tr>
<td>simple-i-dispatch-callback-object</td>
<td>126</td>
</tr>
<tr>
<td>standard-automation-collection</td>
<td>127</td>
</tr>
<tr>
<td>standard-i-connection-point-container</td>
<td>128</td>
</tr>
<tr>
<td>standard-i-dispatch</td>
<td>129</td>
</tr>
<tr>
<td>with-coclass</td>
<td>130</td>
</tr>
</tbody>
</table>
5 Tools 133

5.1 The COM Implementation Browser 133
5.2 The COM Object Browser 135
5.3 The COM Interface Browser 135
5.4 Editor extensions 136

6 Self-contained examples 137

6.1 Argument passing 137
6.2 Aggregation 137
6.3 OLE embedding of external components 137
6.4 Building an ActiveX control 138
6.5 OLE automation 138

Index
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 CAPI User Guide and 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", "Method signature", "Superclasses", "Subclasses", "Slots", "Accessors", "Readers", "Compatibility note", "Description", "Notes", "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.
Example files

This manual often refers to example files in the LispWorks library via a Lisp form like this:

\[(example-edit-file "com/automation/events/ie-events")\]

These examples are files in your LispWorks installation under \lib/8-0-0-0/examples/. You can simply evaluate the given form to view the example file.

Example files contain instructions about how to use them at the start of the file.

The examples files are in a read-only directory and therefore you should compile them inside the IDE (by the Editor command Compile Buffer or the toolbar button or by choosing Buffer > Compile from the context menu), so it does not try to write a fasl file.

If you want to manipulate an example file or compile it on the disk rather than in the IDE, then you need first to copy the file elsewhere (most easily by using the Editor command Write File or by choosing File > Save As from the context menu).

A Description of the Contents

The manual is divided into four sections, relating to COM, Automation, graphical tools and example files respectively. The COM and Automation sections each contain a user guide and a reference 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.

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

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

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

5 Tools describes some tools which are available in the LispWorks IDE to help with debugging applications using COM/Automation. Please note that your windows may look different from the illustrations shown. This is because some details are controlled by the theme and version of Microsoft Windows, not by LispWorks itself.

6 Self-contained examples lists the example files which are relevant to the content of this manual and are available in the LispWorks library.
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 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

This section describes how to load COM and generate any FLI definitions needed to use it, and how to build a COM DLL.

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 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.
1 Using COM

Pre converted IDL files

<table>
<thead>
<tr>
<th>IDL file</th>
<th>Part of Lisp module</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNKNWN.IDL</td>
<td>com</td>
</tr>
<tr>
<td>WTYPES.IDL</td>
<td>com</td>
</tr>
<tr>
<td>OAIDL.IDL</td>
<td>automation</td>
</tr>
<tr>
<td>OLEAUTO.IDL</td>
<td>automation</td>
</tr>
<tr>
<td>OCIDL.IDL</td>
<td>automation</td>
</tr>
</tbody>
</table>

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:

```lisp
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 Delivery User Guide for more details.

You can use the function `set-register-server-error-reporter` to report when calls to `DllRegisterServer` or `DllUnregisterServer` fail.

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>
<thead>
<tr>
<th>COM name</th>
<th>Lisp name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IUnknown</td>
<td>i-unknown</td>
</tr>
<tr>
<td>pStr</td>
<td>p-str</td>
</tr>
<tr>
<td>DWORD</td>
<td>dword</td>
</tr>
<tr>
<td>IEnumVARIANT</td>
<td>i-enum-variant</td>
</tr>
</tbody>
</table>

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 Using COM

1.4 Initializing the COM runtime

Before you can interact with COM, you must initialize the COM runtime by calling `co-initialize`. This must be called in every thread that uses COM. LispWorks takes care of cleaning up the COM runtime when a thread exits, but you can also do this explicitly using `co-uninitialize`.

1.5 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. Note that you must initialize the COM runtime before calling `create-instance` (see 1.4 Initializing the COM runtime).

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

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

1.6 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++).

1.7 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 1.3 The mapping from COM names to Lisp symbols.

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.

1.8 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 1.3 The mapping from COM names to Lisp symbols and the COM interface pointer is a foreign pointer of type `com-interface`. In both macros, the `args` and `values` are as specified in the 1.8.1 Data conversion when calling COM methods.

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:

```lisp
(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))
```

### 1.8.1 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:

```lisp
(example-edit-file "com/manual/args/args-calling")
```

#### 1.8.1.1 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:

```lisp
import "unkwn.idl";
[
  object,
  uuid(E37A70A0-EFC9-11D5-BF02-000347024BE1)
]
```


```c
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:

```lisp
(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:

```lisp
(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.

### 1.8.1.2 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 equivalent 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:

```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:

```lisp
(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:

```lisp
(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:

```lisp
(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)))
```
1 Using COM

(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))))

1.8.1.3 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 in 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:

```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)
)

1.8.2 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
1.9 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 shown in The relationship between an Lisp object and its COM interface pointers 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.

1.9.1 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 1.2.2 Generating FLI definitions from COM definitions.
2. A COM object class defined with the macro `define-com-implementation`, specifying the interface(s) to implement.
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.
1.9.2 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.

The lifecycle 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 1.9.3 Class factories) 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 lifecycle is the normal Lisp garbage collection process, which removes the object from the Lisp world.
memory when there are no more references to it.

### 1.9.3 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 [3 Using Automation](#) and [4 Automation Reference Entries](#), class factories are created and registered automatically by the `define-automation-component` macro if appropriate.

### 1.9.4 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 [1.9.5 Inheritance](#)). 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.

### 1.9.5 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 [1.9.4 Unimplemented methods](#).

#### 1.9.5.1 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
// 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`:

```lisp
(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.

1.9.5.2 A second example of multiple inheritance

Here is a further extension to the example in 1.9.5.1 An example of multiple inheritance, 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))
Using COM

1. Using COM

   (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 1.9.5.1 An example of multiple inheritance 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.

### 1.9.6 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:

   (example-edit-file "com/manual/args/args-impl")

#### 1.9.6.1 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.

If there is a parameter marked with the vararg attribute then the value must be an array.

### 1.9.6.2 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 Automation types, VT codes and their corresponding Lisp types).
- 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 Automation types, VT codes and their corresponding Lisp types.
- 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 1.9.6.1 FLI types.

### 1.9.6.3 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().
1.9.6.4 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 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 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 Automation types, VT codes and their corresponding Lisp types). 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 Automation types, VT codes and their corresponding Lisp types). 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 Automation types, VT codes and their corresponding Lisp types. 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 Automation types, VT codes and their corresponding Lisp types.

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

1.10 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 1.3 The mapping from COM names to Lisp symbols and the arguments and values are as specified below in 1.10.1 Data conversion when calling COM object.
methods. 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.

1.10.1 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.

1.10.1.1 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.

1.10.1.2 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.

1.10.1.3 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

This chapter documents COM functionality.

**add-ref**

*Function*

**Summary**

Increments the reference count of a COM interface pointer.

**Package**

com

**Signature**

\texttt{add-ref} \ interface-ptr => ref-count

**Arguments**

\texttt{interface-ptr} \ A COM interface pointer.

**Values**

\texttt{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 \texttt{add-ref} should be called whenever an extra reference to \texttt{interface-ptr} is being made. The function invokes the COM method \texttt{IUnknown::AddRef} so the form \texttt{(add-ref ptr)} is equivalent to using \texttt{call-com-interface} as follows:

\[
\text{(call-com-interface \ (ptr \ i-unknown \ add-ref))}
\]

**Examples**

\[
\text{(add-ref \ p-bar)}
\]

**See also**

release
interface-ref
query-interface
call-com-interface
Summary
Reports what action was specified for the automation server.

Package
com

Signature
automation-server-command-line-action => action

Values
action One of the keywords :register, :unregister or :embedding, or nil.

Description
The function automation-server-command-line-action inspects the command line to see what action was specified for the automation server. The possible return values have the following meanings:

:register The server should register itself (by register-server). Specified by /RegServer.
:unregister The server should unregister itself (by unregister-server). Specified by /UnRegServer.
:embedding The server was run with /Embedding or -Embedding.
nil No recognized action.

See also
register-server
unregister-server

Summary
For use as the main function for an automation server.

Package
com

Signature
automation-server-main &key exit-delay exit-function new-process force-server forced-exit-delay quit-on-registry-error handle-registry-error
Arguments

- **exit-delay**: A non-negative real number.
- **exit-function**: A function specifier.
- **new-process**: A boolean.
- **force-server**: A boolean.
- **forced-exit-delay**: A non-negative real number.
- **quit-on-registry-error**: A boolean.
- **handle-registry-error**: A boolean.

Description

The function `automation-server-main` is for use as the main function for an automation server.

**exit-delay**, if supplied, sets the exit delay for `automation-server-top-loop`, by calling `set-automation-server-exit-delay` with it.

**exit-function** is an exit-function for `automation-server-top-loop`. The default value of **exit-function** is `server-can-exit-p`.

**new-process** controls whether to run `automation-server-top-loop` in its own process.

**force-server** controls whether to force running the automation server even if the application starts normally. The default value of **force-server** is t.

**forced-exit-delay** specifies a value for **exit-delay** in seconds when **force-server** is non-nil.

`automation-server-main` checks the command line (using `automation-server-command-line-action`) for what action it should do, and then does it.

If the action is `:register` or `:unregister`, `automation-server-main` tries register or unregister the server (using `register-server` and `unregister-server`). If the operation succeeds, `automation-server-main` just returns `:register` or `:unregister`.

**handle-registry-error** controls what happens if there is an error while trying to register or unregister. If nil is supplied then `error` is called, and if a non-nil value is supplied, then the error is handled. If **handle-registry-error** is not supplied, by default the error is handled, but if the command line contains `-debug` or `/debug`, the error is not handled. The default value of **handle-registry-error** is nil.

**quit-on-registry-error** controls what happens if an error occurs during registration. If it is non-nil (the default), then `automation-server-main` calls `quit` with the appropriate status value (5). Otherwise it returns `:register-failed` or `:unregister-failed`. The default value of **quit-on-registry-error** is t.

If the command line action is `:embedding` or the action is nil and **force-server** is non-nil (the default) then `automation-server-main` runs the server by using `automation-server-top-loop`. If **new-process** is nil (the default), `automation-server-top-loop` is called on the current process. In this case `automation-server-main` returns only after `automation-server-top-loop` exits (and the server was closed). If **new-process** is true, `automation-server-top-loop` is called on its own process and `automation-server-main` returns immediately.

If the server is "forced", that is the action is nil but **force-server** is non-nil, and **forced-exit-delay** is non-nil, **exit-delay** is set to **forced-exit-delay** (using `set-automation-server-exit-delay`). This overrides the supplied for **exit-delay**.
automation-server-main returns the result of automation-server-command-line-action, except in the case of registry failure as described above.

Notes

1. automation-server-main is intended to be used as the main function in an automation server that is delivered as an executable (rather than as a DLL).

2. When the application acts only as automation server, automation-server-main can be the function argument to deliver, or the restart-function in save-image (multiprocessing t is needed too). It will deal correctly with registration when the command line argument is supplied, otherwise runs the server until it can exit and then returns (the application will exit because there will not be any other processes).

3. When the application also needs to do other things, automation-server-main can be used to run the server. Note that with the default values when automation-server-main runs the server it does not return until the server exits, so you need to either pass :new-process t, or run it on its own process. You will also need to consider whether to wait when failing to register, and hence may want to pass :quit-on-registry-failure nil.

See also

automation-server-top-loop
automation-server-command-line-action
set-automation-server-exit-delay

---

### automation-server-top-loop

**Function**

**Summary**

A function to run a COM server.

**Package**

com

**Signature**

automation-server-top-loop &key exit-delay exit-function

**Arguments**

- **exit-delay**
  - A non-negative real number specifying a time in seconds.

- **exit-function**
  - A function designator.

**Description**

The function automation-server-top-loop calls co-initialize and start-factories, and then processes messages, until the server can exit. Since COM works by messages, it will end up processing all COM requests.

exit-function determines when the server can exit. It defaults to server-can-exit-p, which is normally the right function. This returns t when the COM server is not used and there are no other "working processes". See the documentation for server-can-exit-p. When exit-function is supplied, it needs to be a function of no arguments which returns true when the server can exit. exit-function is used like a wait function: it is called repeatedly, it needs to be reasonably fast, and should not wait for anything.
Once the server can exit, `automation-server-top-loop` delays exiting for another period of time, `exit-delay` seconds. `exit-delay` defaults to 5, and can be set by calling `set-automation-server-exit-delay`. If supplied, `exit-delay` is passed to `set-automation-server-exit-delay` on entry. However, later calls to `set-automation-server-exit-delay` can change the exit delay.

After the delay `automation-server-top-loop` checks again by calling `exit-function`. If this returns false it goes on to process messages. Otherwise it stops the factories, calls `co-uninitialize` and returns.

Notes

1. `automation-server-top-loop` interacts with the `deliver` keyword `:quit-when-no-windows`, such that the delivered application does not `quit` even after all CAPI windows are closed as long as `automation-server-top-loop` has not returned.

2. `automation-server-top-loop` does not return while the server is active. Typically it will be running on its own process.

3. `automation-server-top-loop` uses `mp:general-handle-event` to process Lisp events, so it is possible to run in the same thread operations that rely on such messages. In particular, CAPI windows can start on the same process. However, all COM input is processed in this thread, so it is probably better to start CAPI windows on other processes, so that they do not interfere with each other.

4. `automation-server-top-loop` does not return a useful value.

See also

`start-factories`  
`stop-factories`  
`automation-server-main`  
`server-can-exit`  
`set-automation-server-exit-delay`

---

**call-com-interface**

*Macro*

**Summary**

Invokes a method from a particular COM interface.

**Package**

`com`

**Signature**

`call-com-interface spec (arg)* => value*`

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

**Arguments**

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

`arg`  
Arguments to the method (see 1.8.1 Data conversion when calling COM methods for details).

`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.

Values

value*
Values from the method (see \texttt{1.8.1 Data conversion when calling COM methods} for details).

Description

The macro \texttt{call-com-interface} invokes the method \texttt{method-name} for the COM interface \texttt{interface-name}, which should the type or a supertype of the actual type of \texttt{interface-ptr}. \texttt{args} and \texttt{value*} are described in detail in \texttt{1.8.1 Data conversion when calling COM methods}.

Examples

This example invokes the COM method \texttt{GetTypeInfo} in the interface \texttt{IDispatch}.

\begin{verbatim}
(defun get-type-info (disp tinfo &key
   (locale \texttt{LOCALE_SYSTEM_DEFAULT}))
   (multiple-value-bind (hres typeinfo)
     (call-com-interface
      (disp \texttt{i-dispatch get-type-info})
      tinfo locale)
     (check-hresult hres \texttt{'get-type-info})
     typeinfo))
\end{verbatim}

See also

\texttt{with-com-interface}

\texttt{query-interface}

\texttt{add-ref}

\texttt{release}

\texttt{call-com-object} \hspace{1em} \textit{Macro}

Summary

Invokes a COM method on a COM object.

Package

\texttt{com}

Signature

\texttt{call-com-object} \texttt{spec {arg} => value*}

\texttt{spec ::= (object class-name method-spec &key interface)}

\texttt{method-spec ::= method-name | (interface-name method-name)}
Arguments

*spec*

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

*arg*↓

Arguments to the method (see §10.1 Data conversion when calling COM object methods for details).

*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.

*interface*↓

A form.

*method-name*↓

A symbol naming the method to call.

*interface-name*↓

A symbol.

Values

*value*↓

Values from the method (see §10.1 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`. `args` and `value*` are described in detail in §10.1 Data conversion when calling COM object methods.

If `method-spec` contains an `interface-name`, then it should name 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`.

If `interface` is supplied, it should be a form that, when evaluated, yields a COM interface pointer. This is only needed if the definition of the method being called has the `interface` keyword in its `class-spec`.

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

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 macro check-hresult checks hresult and returns if it is one of the 'succeeded' values, for instance S_OK or S_FALSE. Otherwise check-hresult signals an error of type com-error, which will include 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-create-guid

Summary
Makes a unique refguid object.

Package
com
Signature

\texttt{co-create-guid \&key register => refguid}

Arguments

\texttt{register} \downarrow \quad \text{A generalized boolean.}

Values

\texttt{refguid} \quad \text{A \texttt{refguid} object.}

Description

The function \texttt{co-create-guid} makes a new unique \texttt{refguid} object. If \texttt{register} is true (the default), then the table of known refguids is updated.

Examples

Make a GUID without registering it in the table of known refguids:

\begin{verbatim}
    (com:co-create-guid :register nil)
\end{verbatim}

\begin{verbatim}
=>
#<REFGUID FOO C76B64AF-969A-4EFF-97BC-6CE2EB65019B>
\end{verbatim}

See also

\texttt{refguid}
\texttt{make-guid-from-string}
\texttt{com-interface-refguid}
\texttt{guid-equal}
\texttt{guid-to-string}
\texttt{refguid-interface-name}

---

\textbf{co-initialize} \quad \textit{Function}

Summary

Initialize the COM library in the current thread.

Package

\texttt{com}

Signature

\texttt{co-initialize \&optional co-init}

Arguments

\texttt{co-init} \downarrow \quad \text{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`.

LispWorks takes care of cleaning up COM when a thread exits, but you can also do this explicitly using **co-uninitialize**.

Examples

```
(co-initialize)
```

See also

**co-uninitialize**

### com-error

**Condition Class**

Summary

The condition class used to signal errors from COM.

Package

`com`

Superclasses

`cl:error`

Subclasses

`com-dispatch-involve-exception-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 condition class **com-error** is used by the Lisp COM API when signaling errors that originate as `hresult` code from COM.
Examples

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**

*System Class*

**Summary**

The class of all COM interface pointers.

**Package**

com

**Superclasses**

t

**Description**

The system class `com-interface` is used for all COM interface pointers.

**Examples**

```
(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 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 1.2.2 Generating FLI definitions from COM definitions or 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
cl: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.

Examples
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

Method signatures
com-object-destructor (object standard-i-unknown)
com-object-destructor :around (object standard-i-unknown)

Arguments
object↓ A COM object.
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 `object`, i.e. where the overall reference count becomes zero. After this, `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

```lisp
(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.

**Examples**

```lisp
(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*

**Method signatures**

com-object INITIALIZE (*object* standard-i-unknown)

**Arguments**

object

A COM object.

**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 *object*. Prior to this, *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**

```lisp
(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**.
Method signatures

\texttt{com-object-query-interface (object standard-i-unknown) (iid t)}

Arguments

- \texttt{object} \downarrow: A COM object.
- \texttt{iid} \downarrow: A GUID foreign pointer.

Values

- \texttt{interface-for-iid} \downarrow: The new interface pointer or \texttt{nil} if none.
- \texttt{skip-add-ref-p} \downarrow: A boolean.

Description

The generic function \texttt{com-object-query-interface} is called by the built-in implementation of \texttt{query-interface} for the class \texttt{standard-i-unknown}. \texttt{iid} is the GUID of the interface to return.

If \texttt{skip-add-ref-p} is \texttt{nil} then \texttt{query-interface} will invoke the COM method \texttt{IUnknown::AddRef} on \texttt{interface-for-iid} before returning it.

The built-in primary method specializing on \texttt{standard-i-unknown} handles the \texttt{i-unknown} interface and all the interfaces specified by the \texttt{define-com-implementation} form for the class of \texttt{object}.

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

You should not call \texttt{com-object-query-interface} directly.

See also

- \texttt{define-com-implementation}
- \texttt{standard-i-unknown}

\textbf{co-task-mem-alloc} \quad \textbf{Function}

Summary

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

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

Arguments

- **type**
  
  A foreign type.

- **pointer-type**
  
  A foreign pointer type.

- **initial-element**
  
  An object.

- **initial-contents**
  
  A list.

- **nelems**
  
  An integer.

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. `type`, `pointer-type`, `initial-element`, `initial-contents` and `nelems` are handled in the same way as for the function `fli:allocate-foreign-object`.

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
```
2 COM Reference Entries

Arguments

\( \text{pointer} \)  
A foreign pointer for the block to be freed.

Values

\( \text{pointer2} \)  
The same as \text{pointer}.

Description

The function \text{co-task-mem-free} calls the C function \text{CoTaskMemFree()} to free a block of memory pointed to by \text{pointer}. \text{pointer} should not be dereferenced after calling this function.

Examples

\[(\text{co-task-mem-free ptr})\]

See also

\text{co-task-mem-alloc}

---

\text{co-uninitialize}  
Function

Summary

Close the COM library in the current thread.

Package

\text{com}

Signature

\text{co-uninitialize}

Description

The function \text{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

\[(\text{co-uninitialize})\]

See also

\text{co-initialize}
create-instance

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 value from the CLSCTX enumeration.

riid⇓
An optional refiid giving the name of the COM interface return.

errorp⇓
A boolean. The default is t.

Values

interface.ptr A COM interface pointer for riid.

Description
The function create-instance creates an instance of the COM server associated with clsid and returns an interface pointer for its riid interface. If riid is nil, then i-unknown is used.

If the server cannot be started, then an error of type com-error will be signaled 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.

clsctx indicate the execution contexts in which an object is to be run. It defaults to CLSCTX_SERVER.

Notes
You must initialize the COM runtime before calling create-instance (see 1.4 Initializing the COM runtime).

To create an i-dispatch interface and set an event handler, you can use create-instance-with-events.

Examples

(create-instance
 "000209FF-0000-0000-C000-000000000046")
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 named class-name, 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).

slot-specifiers are standard defclass slot definitions.

class-options are standard defclass options. In addition the following class-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 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 *from-class-name*, which 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`:

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

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

```lisp
(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-names* (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 subinterface `i-foo-public`, the following definition:

```lisp
(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:

```lisp
(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

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

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

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

Package
`com`

Signature

```lisp
(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.
- `class-spec` Specifies the implementation class and variables bound to the object within forms.
- `arg-spec` Describes one of the method's arguments.
- `form` Forms which implement the method. The value of the final form is returned as the result of the method.
- `method-name` A symbol naming the method to define.
- `interface-name` A symbol.
- `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.
- `parameter-name` A symbol which will be bound to that argument's value while 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.

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 containing `interface-name` is required if `class-name`...
implements more than one interface with a method called \textit{method-name} (analogous to the C++ syntax \texttt{InterfaceName::MethodName}).

When the COM method is called, each \textit{form} is evaluated in a lexical environment containing the following bindings.

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

If present, the symbol \texttt{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 its \textit{direction} and \textit{pass-style} and the corresponding \textit{parameter-name} is bound to the converted value. See \texttt{1.9.6 Data conversion in define-com-method} for details.

The value of the final \textit{form} should be an \texttt{HRESULT}, which is returned from the COM method.

If an error is to be returned from an Automation method, the function \texttt{set-error-info} can be used to provide more details to the caller.

\textbf{Examples}

\begin{verbatim}
(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)
\end{verbatim}

See also

\texttt{define-com-implementation}
\texttt{set-error-info}
\texttt{set-variant}

---

\textbf{find-clsid}

\textit{Function}

\textbf{Summary}

Searches the registry for a GUID or ProgId.

\textbf{Package}

\texttt{com}

\textbf{Signature}

\begin{verbatim}
find-clsid name &optional errorp => refguid
\end{verbatim}

\textbf{Arguments}

\begin{verbatim}
name\Downarrow \hspace{1cm} \text{A string or a refguid.}
errorp\Downarrow \hspace{1cm} \text{A generalized boolean.}
\end{verbatim}
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`.

Examples

To find the GUID of the Explorer ActiveX:

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

get-object

Function

Summary

Returns an interface pointer for a named object.

Package

com

Signature

generic name &key riid errorp => interface-ptr

Arguments

name  A string.

riid  An optional `refiid` giving the name of the COM interface return.

errorp  A boolean. The default value is `t`.

Values

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

Description

The function `get-object` finds an existing object named by `name` in the Running Object Table or activates the object if it is not running.

`get-object` returns an interface pointer for the object's `riid` interface. If `riid` is `nil`, then `i-unknown` is used.

If an error occurs, an error of type `com-error` will be signaled if `errorp` is non-nil, otherwise `nil` will be returned.
2 COM Reference Entries

Examples

If C:\temp\spreadsheet.xls is open in Microsoft Excel 2007, then its WorkBook interface can be obtained using:

```plaintext
(get-object "c:\Temp\spreadsheet.xls"
 :riid 'i-dispatch)
```

See also

create-instance
create-object
get-active-object

---

**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**

```plaintext
(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 `guid` to a string of hex characters in the standard-format.

**Examples**

```
(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`

**Syntax**

`hresult`

**Description**

The FLI type `hresult` 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.

Examples

E_NOTIMPL is negative, so:

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

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

See also

hresult
cHECK-hresult
com-error

### interface-ref

**Accessor**

**Summary**

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

**Package**

com

**Signature**

```
interface-ref iptr-place => iptr
```

```
setf (interface-ref iptr-place) iptr => iptr
```

**Arguments**

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

/iptr/  
A COM interface pointer or nil.

**Values**

/iptr/  
A COM interface pointer or nil.

**Description**

The accessor interface-ref is useful when manipulating a place containing an interface pointer.

The setf form increments the reference count, as if by add-ref, of iptr, unless it is nil. It then decrements the reference count, as if by release, of the existing value in iptr-place, 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-place is set to iptr.

The reader interface-ref simply returns the interface pointer stored in iptr-place and does no reference counting. It may be useful in a form which both reads and writes a place like incf.
2 COM Reference Entries

See also

add-ref
release

i-unknown

COM Interface Type

Summary

The Lisp name for the IUnknown COM interface.

Package

com

Description

The COM interface type i-unknown is the name given to the IUnknown COM interface within Lisp. The name results from the standard mapping described in 1.3 The mapping from COM names to Lisp symbols.

Examples

(query-interface ptr 'i-unknown)

See also

standard-i-unknown
i-dispatch

make-factory-entry

Function

Summary

Makes 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.
**2 COM Reference Entries**

*constructor-extra-args*

Extra arguments to pass to *constructor-function*.

*friendly-name*

A string.

*prog-id*

A string.

*version-independent-prog-id*

A string.

**Description**

The function `make-factory-entry` makes an object to contain all the information for class factory registration in the COM runtime for *clsid*. 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 3 Using Automation.

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

If `constructor-function` is non-nil, it is called by LispWorks with the unknown-outer (non-nil if *aggregation* is being used), the IID of the interface to return and the values in `constructor-extra-args`. It should return three values: the `hresult`, the COM interface pointer and the instance of *implementation-name*.

`constructor-extra-args` supplies extra arguments to pass to `constructor-function`. It defaults to a list containing *implementation-name*.

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

*prog-id* and *version-independent-prog-id* specify the ProgID and VersionIndependentProgID of the coclass when it is registered.

**Examples**

```
(make-factory-entry
 :clsid (make-guid-from-string #x"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`
2 COM Reference Entries

Signature

\texttt{make-guid-from-string string &optional interface-name \Rightarrow refguid}

Arguments

\texttt{string} A string in the standard hex format for GUIDs.

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

Values

\texttt{refguid} A \texttt{refguid} object matching \texttt{string}.

Description

The function \texttt{make-guid-from-string} makes a \texttt{refguid} object from \texttt{string}. If the GUID data matches a known \texttt{refguid}, then that is returned. Otherwise, a new \texttt{refguid} is created and returned. If \texttt{interface-name} is non-nil, then the table of known \texttt{refguids} is updated. If the GUID is already known under a different name, an error is signaled.

Examples

This GUID is a predefined one for \texttt{i-unknown}:

\begin{verbatim}
  (refguid-interface-name
   (make-guid-from-string
    "00000000-0000-0000-C000-000000000046")
  )
  => I-UNKNOWN
\end{verbatim}

See also

\texttt{refguid}

\texttt{com-interface-refguid}

\texttt{guid-equal}

\texttt{guid-to-string}

\texttt{refguid-interface-name}

---

\texttt{midl}

\texttt{Function}

Summary

Converts an IDL file into Lisp FLI definitions.

Package

\texttt{com}

Signature

\texttt{midl file &key package depth mapping-options output-file load import-search-path}
Arguments

- file: A pathname designator.
- package: A package designator.
- depth: A non-negative integer.
- mapping-options: An alist.
- output-file: nil, t or a pathname designator.
- load: A generalized boolean.
- import-search-path: A list of pathname designators or :default.

Description

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

`package` specifies the package in which definitions are created. It defaults to the current package.

`depth` specifies 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 1.2.3 Standard IDL files).

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

If `output-file` 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.

If `load` 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 paths

When the file that `midl` processes contains import statements (which is the normal case, because at least "unknwn.idl" is needed), `midl` looks for the imported file in these directories:

1. A directory in `import-search-path`, or if it is :default in the directory of `file`.

   **Note:** you can pass `import-search-path` as `nil` to prevent searching in the directory of `file`. In many cases that is the more useful behavior.

2. The directories in the list that was set by `midl-set-import-paths`, or if it is :default the directories in the INCLUDE environment variable.

3. The directories in the list that is returned by `midl-default-import-paths`.

The recommended way of getting the standard files to import is to install Windows SDK from microsoft.com. If you install it in the default place, `midl-default-import-paths` should be able to find the right paths. Thus normally installing the Windows SDK is all you need to do to get the standard midl files.

Notes

`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-default-import-paths

**Function**

**Summary**

Returns the default directories for `midl` to search for imported idl files.

**Package**

`com`

**Signature**

```
midl-default-import-paths => paths-list
```

**Values**

```
paths-list A list.
```

**Description**

The function `midl-default-import-paths` returns the default directories for `midl` to search for imported idl files. See `midl` for more details.

You can call `midl-default-import-paths` to see what paths `midl` is going to use. Microsoft do not actually document where you should be looking for imported files, so there is an element of guessing in `midl-default-import-paths`, but if you install the Windows SDK in the default place it should work.

If the Windows SDK is not installed, `midl-default-import-paths` tries to see if the PlatformSDK (the previous incarnation of the Windows SDK) is installed, and uses it instead.

See also

`midl`
Summary
Used to include IDL files in a Lisp system definition.

Package
com

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

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

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

See also
midl

midl-set-import-paths

Summary
Sets an internal list for midl to search for imported files.

Package
com

Signature

```
midl-set-import-paths paths-list
```

Arguments

```
paths-list
```

A list of path-specs (see below), a single path-spec or the keyword :default.

Description

The function midl-set-import-paths sets an internal list for midl to search for imported files. This list overrides the value of the INCLUDE environment variable.
paths-list can be either a list of path-specs, where a path-spec is either a pathname or a string, or a single path-spec, which is interpreted as a list of this path-spec. It can also be the keyword :default, which resets it so it uses the INCLUDE environment variable.

Notes

In most cases midl should be able to find the imported files in the list that is returned by midl-default-import-paths, so midl-set-import-paths should rarely be useful.

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 iid from interface-ptr.

iid can be a symbol naming a COM interface or a refguid foreign pointer containing its iid.

If 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 signaled.

Examples

(query-interface p-foo 'i-bar)
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 ⇓  A COM class name.
object ⇓  A COM object to be queried.
 iid ⇓  The iid of a COM interface.
 ppv-object ⇓  A foreign pointer or nil.

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 iid from object.

iid can be a symbol naming a COM interface or a refguid foreign pointer containing its iid.

class-name must be the COM object class name of object or one of its superclass names.

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. If ppv-object is non-nil, then interface-ptr-for-iid will be stored there as well.

Examples

(query-object-interface foo-impl p-foo 'i-bar)
See also

refguid
hrresult

refguid

Summary

A FLI type used to refer to GUID objects.

Package

com

Syntax

refguid

Description

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

Examples

  (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

Summary

Returns the COM interface name of a refguid if known.

Package

com
2 COM Reference Entries

Signature

reffguid-interface-name refguid => interface-name

Arguments

reffguid

A refguid object.

Values

interface-name

A symbol naming the COM interface of refguid.

Description

The function refguid-interface-name returns a symbol naming the COM interface of refguid, which must be a refguid object known to Lisp.

Examples

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

See also

reffguid
com-interface-reffguid
guid-equal
guid-to-string
make-guid-from-string

---

reffid

FLI Type Descriptor

Summary

A FLI type used to refer to iids.

Package

com

Syntax

reffid

Description

The FLI type reffid 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-reffguid. Otherwise the value should be a foreign pointer to a GUID structure, which is passed directly without conversion.
Examples

Given the definition of `print-iid`:

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

then these two forms are equivalent:

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

```lisp
(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**

```lisp
register-class-factory-entry new-factory-entry
```

**Arguments**

`new-factory-entry`  
A factory entry from `make-factory-entry`.

**Description**

The function `register-class-factory-entry` registers `new-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 3 Using Automation.

**Examples**

See also

`make-factory-entry`

`start-factories`

`stop-factories`

`register-server`

`unregister-server`
register-server

Summary
Externally registers all class factories known to Lisp.

Package
com

Signature
register-server &key clsetx

Arguments
clsetx
A value from the CLSCTX enumeration.

Description
The function register-server 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 wherever the system definition specified, but after using LispWorks delivery it must be located in the directory containing the application's executable or DLL.

register-server should be called when an application is installed, usually by detecting the /RegServer command line argument.

clsetx indicates the execution contexts in which class factories should be used. It defaults to CLSCTX_INPROC_SERVER.

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.

Examples

(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
**release**

**Function**

**Summary**
Decrement 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 `interface-ptr` 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))
```

**Examples**

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

**See also**

add-ref
interface-ref
query-interface
with-temp-interface

**server-can-exit-p**

**Server-in-use-p**

**Functions**

**Summary**
Predicates for whether a COM server is in use or can exit.
**Package**

com

**Signatures**

```plaintext
server-can-exit-p => result
server-in-use-p => result
```

**Values**

```plaintext
result A boolean.
```

**Description**

The function `server-in-use-p` returns true when the COM server is in use, which means one or more of the following:

1. There are live objects other than the class factories.
2. Any of the class factories has more than one reference.
3. The server is locked by a client call to the COM method `IClassFactory::LockServer`.

The function `server-can-exit-p` returns true if the server can exit, which means that the server is not in use (that is, `(not (server-in-use-p))` returns `t`), and also that there are no other "working processes", which means that all other processes except the one that calls `server-can-exit-p` are "Internal servers" (see `mp:process-run-function`).

The main purpose of `server-can-exit-p` is to be the `exit-function` for `automation-server-top-loop`, either as the default or called from a supplied `exit-function`.

**See also**

`automation-server-top-loop`

---

**set-automation-server-exit-delay**

**Function**

**Summary**

Sets exit delay used by `automation-server-top-loop`.

**Package**

com

**Signature**

```plaintext
set-automation-server-exit-delay exit-delay
```

**Arguments**

```plaintext
exit-delay A non-negative real number specifying a time in seconds.
```
Description

The function `set-automation-server-exit-delay` sets `exit-delay` as the exit delay used by `automation-server-top-loop` to delay exiting once the server is unused.

The function `set-automation-server-exit-delay` can be called both before and after `automation-server-top-loop`, and can be used repeatedly after `automation-server-top-loop` was called to dynamically change the exit delay. The setting persists over saving and delivering an image, so it can be used in the delivery script too.

See also

`automation-server-top-loop`

---

**set-register-server-error-reporter**  
*Function*

Summary

Allows control over the reporting, logging or debugging of failures from `register-server` and `unregister-server`.

Package

`com`

Signature

`set-register-server-error-reporter func => func`

Arguments

`func`  
A function or a fbound symbol.

Values

`func`  
A function or a fbound symbol.

Description

The function `set-register-server-error-reporter` sets up a function `func` that is called to report when calls and automatic calls to `register-server` or `unregister-server` via the system-defined entry points of a DLL fail.

`func` should be a function of two arguments.

The automatic calls happen when registering/unregistering a LispWorks DLL that was saved or delivered with the keyword :com in its :dll-exports (see 1.2.4 Making a COM DLL with LispWorks). If such a call fails, `func` is invoked with the name of the function that failed (currently either `register-server` or `unregister-server`) and the condition. `func` should report the failure in a useful way, which would normally mean logging it in a place where you can inspect it later.

Notes

1. After `func` returns or throws out, the automatic call returns with an appropriate failure code, and the code that tries to register (that is, the program that called `DllRegisterServer` or `DllUnregisterServer`) should normally print an error too. For example, `regsvr32` would raise a dialog by default. However, this dialog will not contain any information about what failed inside Lisp.
2. By default (that is, if you do not call \texttt{set-register-server-error-reporter}) any such error is simply printed to standard output.

3. \texttt{func} can force entering the debugger using \texttt{cl:invoke-debugger}, which may sometimes be useful during development.

See also

\texttt{register-server}
\texttt{unregister-server}

\texttt{s\_ok}

\textbf{Macro}

\textbf{Summary}

Compares a result code to the value of \texttt{S\_OK}.

\textbf{Package}

\texttt{com}

\textbf{Signature}

\texttt{s\_ok hresult => flag}

\textbf{Arguments}

\texttt{hresult} \hspace{0.5cm} An integer \texttt{hresult}.

\textbf{Values}

\texttt{flag} \hspace{0.5cm} A boolean.

\textbf{Description}

The macro \texttt{s\_ok} checks \texttt{hresult} and returns true if its value is that of the constant \texttt{S\_OK}. Otherwise it returns false.

\textbf{Examples}

\begin{verbatim}
(S_OK S_OK) => t
(S_OK S\_FALSE) => nil
(S_OK E\_NOINTERFACE) => nil
\end{verbatim}

See also

\texttt{succeeded}
\texttt{hresult}
\texttt{hresult\_equal}
\texttt{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.

**Examples**

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

```scheme
(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`
**start-factories**

**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 function `start-factories` 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**

**Summary**

Stops all the registered class factories.

**Package**

com

**Signature**

`stop-factories`
2 COM Reference Entries

Description

The function \texttt{stop-factories} 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

\texttt{register-class-factory-entry}  
\texttt{start-factories}  
\texttt{register-server}  
\texttt{unregister-server}  
\texttt{co-uninitialize}

\textbf{succeeded} \hspace{1cm} \textit{Macro}

Summary

Checks an \texttt{hresult} for success.

Package

\texttt{com}

Signature

\texttt{succeeded} \hspace{.1cm} \texttt{hresult => flag}

Arguments

\texttt{hresult} \hspace{1cm} An integer \texttt{hresult}.

Values

\texttt{flag} \hspace{1cm} A boolean.

Description

The macro \texttt{succeeded} checks \texttt{hresult} and returns true if the it is one of the 'succeeded' values, for instance \texttt{S_OK} or \texttt{S_FALSE}. Otherwise, it returns false.

Examples

\begin{verbatim}
  (succeeded S_OK) => t

  (succeeded E_NOINTERFACE) => nil
\end{verbatim}

See also

\texttt{check-hresult}  
\texttt{hresult}  
\texttt{hresult-equal}  
\texttt{s_ok}
unregister-server  

Summary
Externally unregisters all class factories known to Lisp.

Package
com

Signature
unregister-server

Description
The function `unregister-server` 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.

Examples

```lisp
(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
set-register-server-error-reporter
**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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>disp</code></td>
<td>The names of the dispatch function and interface.</td>
</tr>
<tr>
<td><code>interface-ptr</code></td>
<td>A form which is evaluated to yield a COM interface pointer that implements <code>interface-name</code>.</td>
</tr>
<tr>
<td><code>form</code></td>
<td>A form to be evaluated.</td>
</tr>
<tr>
<td><code>dispatch-function</code></td>
<td>A symbol.</td>
</tr>
<tr>
<td><code>interface-name</code></td>
<td>A symbol which names the COM interface. It is not evaluated.</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>values</code></td>
<td>The values returned by the last <code>form</code>.</td>
</tr>
</tbody>
</table>

**Description**

The macro `with-com-interface` evaluates each `form` in a lexical environment where `dispatch-function` is defined as a local macro.

`dispatch-function` can be used to invoke the methods on `interface-ptr` for the COM interface `interface-name`, which should be the type or a supertype of the actual type of `interface-ptr`.

`dispatch-function` 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 1.8.1 *Data conversion when calling COM methods* for details).
- `values` Values from the method (see 1.8.1 *Data conversion when calling COM methods* for details).

**Examples**

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

---

75
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.

object

A form which is evaluated to yield a COM object.

form

A form to be evaluated.

dispatch-function

A symbol.

class-name

A symbol which names the COM implementation class. It is not evaluated.

interface

A form.

Values

values

The values returned by the last form.

Description

The macro with-com-object evaluates each form in a lexical environment where dispatch-function is defined as a local macro.

dispatch-function can be used to invoked the methods on object for the COM class class-name, which should be the type or a supertype of the actual type of object.

dispatch-function has the following signature:

dispatch-function method-spec arg* => values
2 COM Reference Entries

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 1.10.1 Data conversion when calling COM object methods for details).
  values                Values from the method (see 1.10.1 Data conversion when calling COM object methods for details).

If interface is supplied, it should be a form that, when evaluated, yields a COM interface pointer. This is only needed if the definition of the method being called has the :interface keyword in its class-spec.

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.

Examples

(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-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}* => value*

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

Arguments

interface-ptr\n A form which is evaluated to yield a COM interface pointer to query.
form\n A form to be evaluated.
2 COM Reference Entries

punknown
A symbol.

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

errorp
A generalized boolean.

dispatch
A symbol.

Values
value*
The values returned by the last form.

Description
The macro \texttt{with-query-interface} calls \texttt{query-interface} to find an interface pointer for \texttt{interface-name} from the existing COM interface pointer \texttt{interface-ptr}. It evaluates each \texttt{form} with the variable \texttt{punknown} 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 \texttt{errorp} is true, then \texttt{punknown} is bound to \texttt{nil} if the interface pointer cannot be found, otherwise an error of type \texttt{com-error} is signaled.

If \texttt{dispatch} is non-nil, then a local macro named by \texttt{dispatch} is created as if by \texttt{with-com-interface} to invoke COM interface methods on \texttt{punkknown}.

Examples
This example invokes the methods on an \texttt{i-bar} interface pointer queried from an existing interface pointer.

\begin{verbatim}
(with-query-interface (p-bar i-bar
  :dispatch call-bar)
  p-foo
  (call-bar bar-init)
  (call-bar bar-print))
\end{verbatim}

See also
\texttt{query-interface}
\texttt{release}
\texttt{with-temp-interface}

\begin{verbatim}
with-temp-interface
\end{verbatim}
Macro

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

Package
com

Signature
\texttt{with-temp-interface (var) interface-ptr \{form\}* => value*}
2 COM Reference Entries

Arguments

\[ var \]
A symbol.

\[ interface-ptr \]
A form which is evaluated to yield a COM interface pointer.

\[ form \]
A form to be evaluated.

Values

\[ value^* \]
The values returned by the last \[ form \].

Description

The macro \[ with-temp-interface \] evaluates each \[ form \] with the variable \[ var \] 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.

Examples

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

\hspace{1cm} member-id)

\hspace{1.5cm} (with-temp-interface (typeinfo)
\hspace{1.5cm} \hspace{1cm} (get-type-info disp tinfo)
\hspace{1.5cm} \hspace{1cm} (call-com-interface (typeinfo i-type-info
\hspace{2cm} \hspace{1cm} \hspace{1cm} get-documentation)
\hspace{2cm} \hspace{1cm} \hspace{1cm} member-id))))
\]

See also

\[ release \]
\[ with-query-interface \]
3 Using Automation

3.1 Including Automation in a Lisp application

This section describes how to load Automation and generate any FLI definitions needed to use it.

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 loading 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 1.3 The mapping from COM names to Lisp symbols.

Note: this is not required by all the APIs, for example see 3.3.2 Calling Automation methods without a type library and 3.4.2 A simple implementation of a single Automation interface.

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 3.1.2 Generating FLI definitions from COM definitions.

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.
There are three kinds of Automation method, each of which is implemented in a type library. If the type library is not available at compile-time, you can use the approach described in the next section.

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. The function `get-active-object` can be used to look for a registered running instance of a coclass in the system Running Object Table.

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 3.1.2 Generating FLI definitions from COM definitions). 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 methods 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 1.3 The mapping from COM names to Lisp symbols and the COM interface pointer is a foreign pointer of type `com-interface`. In all the macros, the `args` and `values` are as specified in the 3.3.3 Data conversion when calling Automation methods.

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 3.3.1 Calling Automation methods using a type library, 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 methods 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 functions, 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 `args` and `values` are as specified in the 3.3.3 Data conversion when calling Automation methods.

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

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.

Optional parameters can be passed as :not-specified if they are not needed. Alternatively, they can be omitted if all remaining optional arguments are also omitted.

If there is a parameter marked with the `vararg` attribute then any arguments after the last optional argument will be collected...
into an array and passed as the value of that parameter.

### 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 an `MyDocument` interface pointer:

```lisp
(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.

```lisp
(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 Occurred (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 Using Automation

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 1.9 Implementing COM interfaces in Lisp.

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-invokes.

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 3.1 Including Automation in a Lisp application.
2. A COM object class defined with define-automation-component or define-automation-collection, specifying the coclass or interface(s) to implement.
4. For an out-of-process Automation component, either use automation-server-main or have registration code which calls register-server and unregister-server, typically after checking the result of automation-server-command-line-action or explicitly checking the command line for arguments /RegServer and /UnRegServer.
5. Initialization code which either calls automation-server-top-loop or automation-server-main, or 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 files here:

```lisp
(example-edit-file "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:

```lisp
(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 Registering a running object for use by other applications

If other applications need to be able to find one of your running objects from its coclass, then call `register-active-object` to register an interface pointer for the object in the system Running Object Table. Call `revoke-active-object` to remove the registration.

### 3.4.7 Automation of a CAPI application

For an example of how to implement an Automation server that controls a CAPI application, see the file:

```lisp
(example-edit-file "com/automation/capi-application/build")
```
3.5 Examples of using Automation

Several complete examples are provided in the examples subdirectory of your LispWorks library.

A simple Automation application:

```lisp
(example-edit-file "com/automation/capi-application/readme.txt")

(example-edit-file "com/automation/cl-smtp/clsmtp-impl-build")
```

Controlling an Automation application:

```lisp
(example-edit-file "com/automation/capi-application/readme.txt")

(example-edit-file "com/automation/cl-smtp/clsmtp-test")
```

Getting events from COM interfaces:

```lisp
(example-edit-file "com/automation/events/ie-events")

(example-edit-file "com/automation/capi-application/readme.txt")
```
4 Automation Reference Entries

This chapter documents Automation functionality.

**call dispatched-get-property**  
*Macro*

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

**Package**
com

**Signature**

```lisp
(call-dispatch-get-property spec (arg) => value)
```

**Arguments**

- **spec**: The interface pointer and a specification of the method to be called.
- **arg**: Arguments to the method (see 3.3.3 Data conversion when calling Automation methods for details).
- **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.

**Values**

- **value**: Values from the method (see 3.3.3 Data conversion when calling Automation methods for details).

**Description**
The macro `call-dispatch-get-property` is used to invoke an Automation property getter method from Lisp.

`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`, which 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 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.
Examples

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}* => value*

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

Arguments

spec  

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

arg↓

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

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.

Values

value*

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

Description

The macro call-dispatch-method is used to invoke an Automation method from Lisp.

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`, which 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 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.

**Examples**

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

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

See also

---

### call-dispatch-put-property

**Macro**

**Summary**

Calls an Automation property setter method from a particular interface.

**Package**

`com`

**Signature**

```lisp
(call-dispatch-put-property spec (arg)* => value*)
```

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

**Arguments**

- `spec` The interface pointer and a specification of the method to be called.
- `arg` Arguments to the method (see 3.3.3 Data conversion when calling Automation methods for details).
- `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.

**Values**

- `value*` Values from the method (see 3.3.3 Data conversion when calling Automation methods for details).
Description

The macro `call-dispatch-put-property` is used to invoke an Automation property setter method from Lisp. `dispinterface-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`, which 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 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.

Examples

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

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

See also

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

---

### com-dispatch-invoke-exception-error

**Condition Class**

Summary

The condition class used to signal Automation exceptions.

Package

com

Superclasses

com-error

Description

The condition class `com-dispatch-invoke-exception-error` is used by the LispWorks COM API when Automation signals an exception (`DISP_E_EXCEPTION`).

See also

`com-dispatch-invoke-exception-error-info`
**com-dispatch-invoke-exception-error-info**  
*Function*

**Summary**
Retrieves information stored in a `com-dispatch-invoke-exception-error`.

**Package**
`com`

**Signature**
`com-dispatch-invoke-exception-error-info condition fields => field-values`

**Arguments**
- `condition`\↓
  A `com-dispatch-invoke-exception-error`.
- `fields`\↓
  A list of keywords as specified below.

**Values**
- `field-values`\↓
  A list.

**Description**
The function `com-dispatch-invoke-exception-error-info` retrieves information about the exception from `condition`. The keywords in `fields` are used to select which information is returned in `field-values`, which is a list of values corresponding to each keyword in `fields`.

The following keyword are supported in `fields`:
- **:code**
  The error code.
- **:source**
  The source of the error.
- **:description**
  The description of the error.
- **:help-file**
  The help file for the error.
- **:help-context**
  The help context for the error.

**Examples**

```lisp
(handler-case
  (com:invoke-dispatch-method counter "Run")
  (com:com-dispatch-invoke-exception-error (condition)
    (destructuring-bind (code description)
      (com:com-dispatch-invoke-exception-error-info
       condition
       '(:code :description))
      (format *error-output*
       "Run failed with code ~D, description ~S."
       code
       description)))
```


**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.

**Values**

- `value` A value suitable for return from a COM 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`.

`method-name` is a string specifying the name of the method as given by the method declaration in the IDL or type library.

`method-type`, 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 **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 **3.3.3 Data conversion when calling Automation methods**.
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 3.3.3 Data conversion when calling Automation methods. It is ignored for methods that are declared as returning void.

Notes

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.

Examples

(defmethod com:com-object-dispinterface-invoke ((this my-dispinterface) method-name method-type args)
  (cond ((equal method-name "MyProperty")
         (case method-type
               (:get
                (slot-value this 'my-property))
               (:put
                (setf (slot-value this 'my-property)
                      (svref args 0)))))
         ((equal method-name "MyMethod")
          (format t "MyMethod was called~%"))
         (t (call-next-method))))

See also

define-dispinterface-method

create-instance-with-events

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, sinks

Arguments

clsid A string or a refguid giving a CLSID to create.

event-handler A function of four arguments.

args Lisp objects.

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 in `args` except `:event-object`. `clsid` 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

```
(ex example-edit-file "com/automation/events/ie-events")
```

### 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.
4 Automation Reference Entries

Description

The function create-object creates an instance of a coclass and returns its i-dispatch interface pointer. The coclass can be specified directly by supplying clsid or indirectly by supplying progid, which will locate the CLSID from the registry. clsctx indicate the execution contexts in which an object is to be run. It defaults to CLSCTX_SERVER.

Notes

You must initialize the COM runtime before calling create-object (see 1.4 Initializing the COM runtime).

Examples

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

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

```lisp
(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 named by class-name 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-specifiers are standard defclass slot definitions.
class-options 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.

Examples

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 `:coclasm` or `:source-interfaces` options). This provides a complete implementation of the Connection Point protocols.

`slot-specifiers` are standard `defclass` slot definitions.

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

`:coclasm` (\(<\) `coclasm-name`\>)

coclasm-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 `:coclasm` 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 `:coclasm` 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 `:coclasm` 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 `:coclasm` option. This allows the object to implement other interfaces not mentioned in the type library.

`:coclasm-reusable-p` (\(<\) `reusable`\)
If `reusable` is true (the default), then the server running the component can receive requests from more than one application. If `reusable` is `nil`, then the server will receive requests only from the application that started it and the Operating System will start a new instance of the server if required. For more details, see `REGCLS_MULTIPLEUSE` and `REGCLS_SINGLEUSE` in MSDN.

(`:type-library type-library-name)

`type-library-name` is a symbol specifying the name of a type library, mapped from the name given by the "library" statement in the IDL. If this option is specified then an error is signaled if the names used in the`:coclass`, `:interfaces` or `:source-interfaces` class options are not defined by `type-library-name`.

Use `define-com-method`, `define-dispinterface-method` or `com-object-dispinterface-invoke` to define methods in the interfaces implemented by the component. See also 1.9.4 Unimplemented methods.

Examples

```lisp
(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**

Defines a `dispinterface` method.

**Package**

`com`

**Signature**

```lisp
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.
class-spec

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

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.

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.

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.

Values

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 containing interface-name is required if class-name implements more than one interface with a method called method-name (analogous to the C++ syntax InterfaceName::MethodName).

When the method is called, each form is evaluated in a lexical environment containing the following bindings.

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 3.3.3 Data conversion when calling Automation methods. For missing values the value of the parameter is :not-found. For a parameter marked with the vararg attribute, the value will be an array of the values passed in the call. 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 3.3.3 Data conversion when calling Automation methods.

time 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 3.3.3 Data conversion when calling Automation methods. It is ignored for methods that are declared as returning void.

Notes

The define-com-method macro should be used to implement methods in dual interfaces.

See also

define-com-method
com-object-dispinterface-involve
disconnect-standard-sink

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

Summary
Iterates over the items of an Automation Collection.

Package
com

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

Macro
**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 macro **do-collection-items** executes each **form** in turn, with **item** bound to each item of **collection**.

Note that for collections whose items are interface pointers, **forms** must arrange for each pointer to be released when no longer needed.

**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 **3.3 Data conversion when calling Automation methods**.

**Examples**

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

```lisp
(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**.
4 Automation Reference Entries

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.

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.

Examples

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 (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
Signature

\texttt{find-component-value name key-name => result, root}

Arguments

\textit{name} \downarrow \quad \text{A string.}

\textit{key-name} \downarrow \quad \text{A string or a keyword.}

Values

\textit{result} \downarrow \quad \text{A Lisp object.}

\textit{root} \downarrow \quad \text{A keyword.}

Description

The function \texttt{find-component-value} searches the Windows registry for values associated with a component. \textit{name} should be the name of a component (either a ProgID or a GUID).

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

- \texttt{:library} \quad \text{Returns the library that implements the component (if any).}
- \texttt{:inproc-server32} \quad \text{As for \texttt{:library}.}
- \texttt{:local-server32} \quad \text{Returns the executable that implements the component (if any).}
- \texttt{:version} \quad \text{Returns the version.}
- \texttt{:prog-id} \quad \text{Returns the ProgID.}
- \texttt{:version-independent-prog-id} \quad \text{Returns the version-independent ProgId.}
- \texttt{:type-lib} \quad \text{Returns the GUID of the type library.}

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

\textit{find-component-value} simply returns \texttt{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

\begin{verbatim}
  (com:find-component-value "shell.explorer" :version)
\end{verbatim}
get-active-object

Summary

Looks for a registered running instance of a coclass.

Package

com

Signature

get-active-object &key clsid progid riid errorp => interface-ptr

Arguments

clsid⇓ A string or a refguid giving a CLSID to create.
progid⇓ A string giving a ProgID to create.
riid⇓ An optional refiid giving the COM interface name to return.
errorp⇓ A boolean. The default is t.

Values

interface-ptr A COM interface pointer for riid.

Description

The function get-active-object looks for a registered running instance of a coclass in the system Running Object Table and returns its riid interface pointer if any. If riid is nil, then i-unknown is used.

The coclass can be specified directly by supplying clsid or indirectly by supplying progid, which will locate the CLSID from the registry.

If errorp is true, then an error is signaled if no instances are running. Otherwise nil is returned if no instances are running.

Examples

(get-active-object :progid "Excel.Application"
:riid 'i-dispatch)

See also

get-object
get-error-info

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.
fields⇓ A list of keywords specifying the error information fields to return.

Values
field-value* Values corresponding to fields.

Description
The function get-error-info allows the various components of the error information to be retrieved for the last Automation method called. 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.

If errorp is true and an error occurs while retrieving the error information, then an error of type com-error is signaled. Otherwise nil is returned.

Examples

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

See also
set-error-info
**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 `i-dispatch`. That is, it obtains the first return value of `ITypeInfo::GetDocumentation`.

**Examples**

To implement code like this:

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

you would need something like:

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

**See also**

`print-i-dispatch-methods`
`i-dispatch`
`create-object`
`create-instance-with-events`

3.2 Starting a remote Automation server
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.

Notes

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 IDispatch COM interface.
Package
com

Description
The COM interface type \texttt{i-dispatch} is the name given to the \texttt{IDispatch} COM interface within Lisp. The name results from the standard mapping described in \texttt{1.3 The mapping from COM names to Lisp symbols}.

Examples

\begin{verbatim}
   (query-interface ptr 'i-dispatch)
\end{verbatim}

See also
\texttt{i-unknown}
\texttt{standard-i-dispatch}

\begin{tabular}{ll}
\textbf{interface-connect} & \textit{Function} \\
\end{tabular}

\textbf{Summary}
Connects a sink interface pointer to the source of events in another COM interface pointer.

Package
com

\textbf{Signature}
\texttt{interface-connect interface-ptr iid sink-ptr &key errorp => cookie}

\textbf{Arguments}
\begin{itemize}
\item \texttt{interface-ptr} A COM interface pointer that provides the source interface \texttt{iid}. \\
\item \texttt{iid} The iid of the source interface to be connected. The iid can be a symbol naming the interface or a \texttt{refguid} foreign pointer. \\
\item \texttt{sink-ptr} A COM interface that will receive the events for \texttt{iid}. \\
\item \texttt{errorp} A boolean.
\end{itemize}

\textbf{Values}
\begin{itemize}
\item \texttt{cookie} An integer cookie associated with this connection.
\end{itemize}

\textbf{Description}
The function \texttt{interface-connect} connects the COM interface \texttt{sink-ptr} to the connection point in \texttt{interface-ptr} that is named by \texttt{iid}.

If \texttt{errorp} is false, errors connecting \texttt{sink-ptr} will cause \texttt{nil} to be returned. Otherwise an error of type \texttt{com-error} will be signaled.
Examples

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.

**Values**

- `flag`
  A boolean, true for successful disconnection.

**Description**

The function `interface-disconnect` disconnects the connection `cookie` from the connection point in `interface-ptr` that matches `iid`.

If `errorp` is false, errors disconnecting `cookie` will cause `nil` to be returned. Otherwise an error of type `com-error` will be signaled.
Examples

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`

### `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 => value*
```

**Arguments**

- `dispinterface-ptr`<sup>⇓</sup> An Automation interface pointer.
- `name`<sup>⇓</sup> A string or integer.
- `args`<sup>⇓</sup> Arguments passed to the method.

**Values**

- `value*`<sup>⇓</sup> 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.

`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.

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 returned values in value* consist of the primary value of the method (if not void) and the values of any out or in-out parameters. See 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.

Examples

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

```lisp
(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

decl-option

### invoke-dispatch-method

**Function**

**Summary**

Call a dispatch method from an interface pointer.

**Package**

com

**Signature**

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

**Arguments**

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

**Values**

- `value*`: 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.

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.

`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 returned values in `value*` consist of the primary value of the method (if not void) and the values of any `out` or `in-out` parameters. See 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.

Examples

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`

---

### Function

#### invoke-dispatch-put-property

**Summary**

Call a dispatch property setter method from an interface pointer.

**Package**

`com`

**Signature**

```
invoke-dispatch-put-property dispinterface-ptr name &rest args => value*
```

**Arguments**

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

**Values**

- `value*` ↓ Values returned by the method.
4 Automation Reference Entries

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:

```lisp
  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.

`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.

`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 returned values in `value*` consist of the primary value of the method (if not void) and the values of any `out` or `in-out` parameters. See 3.3.3 Data conversion when calling Automation methods for more details.

Examples

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

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

See also

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

---

**lisp-variant**

System Class

Summary

An object that contains a type and a value.

Package

`com`

Superclasses

`t`

Accessors

- `lisp-variant-type`
- `lisp-variant-value`
Description

Instances of the system class `lisp-variant` contains a type and a value, which are described for the function `set-variant`.

See also

`make-lisp-variant`
`set-variant`

---

### 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`

**Arguments**

- `type` `\lla\r
A keyword.

- `value` `\lla\r
A Lisp object.

**Values**

- `lisp-variant` `\lla\r
A `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 defsystem member 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 specify 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 3.1.3 Reducing the size of the converted library.

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 clause like this:
See also

find-component-tlb
:midl-file

print-i-dispatch-methods

Function

Summary

Prints the defined methods for an i-dispatch.

Package

com

Signature

print-i-dispatch-methods  i-dispatch  &optional arguments-p

Arguments

i-dispatch  ⇓
An i-dispatch interface object.

arguments-p  ⇓
A boolean.

Description

The function print-i-dispatch-methods prints the methods that are defined for the i-dispatch.

print-i-dispatch-methods tries to get the information about the methods of i-dispatch and print them in a readable format. If arguments-p is nil then for each each method it prints its name, followed by the invocation type(s) inside curly brackets. Invocation types are:

"Method"  Invoked by invoke-dispatch-method.

"Get"  Invoked by invoke-dispatch-get-property.


If arguments-p is true, print-i-dispatch-methods also prints the types of the arguments for each method. The type of each argument is shown as a plain string followed by the name of the VT_… constant delimited by curly brackets. The type may be preceded by:

By reference  Means the argument has VT_ByRef. The argument in that is passed in Lisp should be the actual type. By reference argument values are returned as multiple values, following the return value of the method if it has one.

Array of  Means it got VT_Array. The argument in Lisp should be an array.

Array of references  Means it got VT_Array and VT_ByRef. The argument needs to be an array of the actual type.
The default value of arguments-p is nil.

Notes

1. **print-i-dispatch-methods** gives only partial information, and is therefore useful only for the simple methods where it is pretty obvious what the arguments are. If the arguments are not obvious, you will need to read the actual documentation.

2. The type Variant means any type, but note that the method may accept only specific types even if the argument is variant.

See also

- get-i-dispatch-name
- i-dispatch
- invoke-dispatch-put-property
- invoke-dispatch-get-property
- invoke-dispatch-method

3.3.2 Calling Automation methods without a type library

---

**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 does not 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 supplying `related-dispatch`. 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 `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`

---

**register-active-object**

**Summary**

Registers an instance of a coclass.

**Package**

`com`

**Signature**

```
register-active-object interface-ptr &key clsid progid flags => token
```

**Arguments**

- `interface-ptr`:
  - A COM interface pointer.
- `clsid`:
  - A string or a `refguid` giving a CLSID to create.
- `progid`:
  - A string giving a ProgID to create.
- `flags`:
  - An integer.

**Values**

- `token`:
  - An integer.

**Description**

The function `register-active-object` registers `interface-ptr` in the system Running Object Table for a specific coclass that the application implements. The coclass can be specified directly by supplying `clsid` or indirectly by supplying `progid`, which will locate the CLSID from the registry.

`flags` can be an integer as specified for the Win32 API function `RegisterActiveObject`. The default value of `flags` is 0.

The returned value `token` can be used with `revoke-active-object` to revoke the registration.
See also

**revoke-active-object**

### revoke-active-object

**Function**

**Summary**

Unregisters a previously registered instance of a coclass.

**Package**

`com`

**Signature**

`revoke-active-object token`

**Arguments**

- `token` An integer.

**Description**

The function `revoke-active-object` revokes the registration of the object associated with `token` in the system Running Object Table. The value of `token` should be one that was returned by a call to `register-active-object`.

See also

**register-active-object**

### 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` a symbol naming a COM interface or a `refguid` foreign pointer.
- `source` A string or `nil`.
- `description` A string or `nil`. 
help-file
A string or nil.

help-context
An integer or nil.

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.

If iid is non-nil, it is set as IID of the interface that defined the error, or nil if none.

If source is non-nil, it is set as the ProgID for the class that raised the error.

If description is non-nil, it is set as the textual description of the error.

If help-file is non-nil, it is set as the path of the help file that describes the error.

If help-context is non-nil, it is set as the help context id for the error.

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 with the following signature:

```
 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.

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-dispatch-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**.

Notes

1. **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.

2. 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**.

3. 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**

```lisp
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.

<table>
<thead>
<tr>
<th>Value of <code>type</code></th>
<th>VT code used</th>
<th>Expected type of <code>value</code></th>
</tr>
</thead>
</table>

123
If type is nil then the actual VT code is chosen dynamically according to the Lisp type of value (see Automation types, VT codes and their corresponding Lisp types).

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 \texttt{i-dispatch} interface.

Package

com

Superclasses

standard-i-dispatch

Initargs

:interface-name The name of the interface to implement. See \texttt{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 \texttt{:method}, \texttt{:get} or \texttt{: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 \texttt{com-object-dispinterfaceinvoke} 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 \texttt{simple-i-dispatch} provides a complete implementation of the \texttt{i-dispatch} interface, without requiring a type library to be parsed. The type information is obtained at run-time when \texttt{query-simple-i-dispatch-interface} is called. The class inherits from \texttt{standard-i-dispatch} to provide the \texttt{i-unknown} interface.

The \texttt{simple-i-dispatch-refguid} reader can be used to return the \texttt{refguid} of the interface. This can only be called after \texttt{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

**Summary**

A generic function that can be implemented to modify the first argument to the *invoke-callback* in `simple-i-dispatch`.

**Package**

`com`

**Signature**

\[
\text{simple-i-dispatch-callback-object} \ (\text{this} \Rightarrow \text{object})
\]

**Method signatures**

\[
\text{simple-i-dispatch-callback-object} \ (\text{this} \ \text{simple-i-dispatch})
\]

**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 the *invoke-callback* of `this`. 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.

**Examples**

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)))

(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

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 Automation types, VT codes and their corresponding Lisp types). The default function is `identity`.

Examples

See the example in this directory:

```
   (example-edit-file "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

```
c
```
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.

Examples

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.

Examples

Given the class definition:

```lisp
(define-automation-component document-impl ()
  ()
  (:coclasse document)
)
```

then:

```lisp
(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}* => value*

disp ::= (dispatch-function coclass-name &key interface-name punk clsctx)
Arguments

disp
The names of the dispatch function, coclass and so on.

form
A form to be evaluated.

dispatch-function
A symbol which will be defined as a macro, as if by \texttt{with-dispatch-interface}. The macro can be used by \texttt{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 \texttt{CLSCTX\_SERVER}.

Values

value*
The values returned by the last \texttt{form}.

Description

The macro \texttt{with-coclass} calls \texttt{create-object} to make an instance of the coclass named by the symbol \texttt{coclass-name}.

If \texttt{interface-name} is supplied then that interface is queried from the component, otherwise the default interface is queried.

Each \texttt{form} is evaluated in turn with \texttt{dispatch-function} bound of a local macro for invoking methods on the interface, as if by \texttt{with-dispatch-interface}. After the forms have been evaluated, the interface pointer is released.

If \texttt{punk} is supplied, it will be bound to the interface pointer while the forms are being evaluated.

\texttt{clsctx} indicate the execution contexts in which an object is to be run. It defaults to \texttt{CLSCTX\_SERVER}.

Examples

If a type library containing the coclass \texttt{TestComponent} has been converted to Lisp, then following can be used to make an instance of component and invoke the \texttt{Greet()} method on the default interface.

\begin{verbatim}
(with-coclass (call-it test-component)
  (call-it greet "hello"))
\end{verbatim}

See also

\texttt{create-object}

\textbf{with-dispatch-interface} \hspace{1cm} \textit{Macro}

Summary

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

Package

\texttt{com}
4 Automation Reference Entries

Signature

\[
\text{with-dispatch-interface} \ disp \ \text{dispinterface-ptr} \ \{\text{form}\}^* \Rightarrow \ \text{value}^*
\]

\[
disp := (\text{dispatch-function} \ \text{dispinterface-name})
\]

Arguments

- \textit{disp}:
  - The names of the dispatch function and Automation interface.
- \textit{dispinterface-ptr}:
  - A form which is evaluated to yield a COM \textit{i-dispatch} interface pointer.
- \textit{form}:
  - A form to be evaluated.
- \textit{dispatch-function}:
  - A symbol which will be defined as a macro, as if by \textit{macrolet}. The macro can be used by \textit{forms} to invoke the methods on \textit{dispinterface-ptr}.
- \textit{dispinterface-name}:
  - A symbol which names the Automation interface. It is not evaluated.

Values

- \textit{value}^*:
  - The values returned by the last \textit{form}.

Description

When the macro \textit{with-dispatch-interface} evaluates \textit{forms}, the local macro \textit{dispatch-function} can be used to invoked the methods for the Automation interface \textit{dispinterface-name}, which should be the type or a supertype of the actual type of the Automation interface pointer \textit{dispinterface-ptr}.

\textit{dispatch-function} has the following signature:

\[
\text{dispatch-function} \ \text{method-name} \ \text{arg}^* \Rightarrow \ \text{values}
\]

where:

- \textit{method-name}:
  - A symbol which names the method. It is not evaluated.
- \textit{arg}:
  - Arguments to the method (see \textit{3.3.3 Data conversion when calling Automation methods} for details).
- \textit{values}:
  - Values from the method (see \textit{3.3.3 Data conversion when calling Automation methods} for details).

Examples

For example, in order to invoke the \texttt{ReFormat} method of a \texttt{MyDocument} interface pointer:

\[
\begin{align*}
\text{(with-dispatch-interface (call-doc my-document) doc} \\
\text{(call-doc re-format))}
\end{align*}
\]

See also

\textit{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.

![COM Implementation Browser](image)
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 1.9.4 Unimplemented methods), yellow means inherited from a superclass (see 1.9.5 Inheritance), 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.

COM Implementation Wizard

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.

**COM Object Browser**

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.
COM Interface Browser

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.
6 Self-contained examples

This chapter enumerates the set of examples in the LispWorks library relevant to the content of this manual. Each example file contains complete, self-contained code and detailed comments, which include one or more entry points near the start of the file which you can run to start the program.

To run the example code:

1. Open the file in the Editor tool in the LispWorks IDE. Evaluating the call to `example-edit-file` shown below will achieve this.

2. In some cases you can simply compile the example code, by `Ctrl+Shift+B`, and then place the cursor at the end of the entry point form and press `Ctrl+X Ctrl+E` to run it. However the comments near the start of the file may have specific instructions, such as how to build a delivered executable or library, so follow these if present.

6.1 Argument passing

These files comprise an example illustrating various argument passing issues when calling and implementing COM methods. To run the example, follow the instructions in `defsys.lisp`.

```lisp
(example-edit-file "com/manual/args/defsys")
(example-edit-file "com/manual/args/args.idl")
(example-edit-file "com/manual/args/args-impl")
(example-edit-file "com/manual/args/args-calling")
```

6.2 Aggregation

These three files contain a simple demonstration of implementing aggregation:

```lisp
(example-edit-file "com/com/aggregation-defsys")
(example-edit-file "com/com/aggregation-idl.idl")
(example-edit-file "com/com/aggregation")
```

6.3 OLE embedding of external components

These examples illustrate OLE embedding of external components in a CAPI interface:

```lisp
(example-edit-file "com/ole/html-viewer")
(example-edit-file "com/ole/flash-player")
```
6.4 Building an ActiveX control

These three files together comprise an example which illustrates building an ActiveX control. Start by reading the comments in the first file:

(example-edit-file "com/ole/control-implementation/deliver")

(example-edit-file "com/ole/control-implementation/defsys")

(example-edit-file "com/ole/control-implementation/my-control-implementation")

This file shows how you can embed the new ActiveX control in another application:

(example-edit-file "com/ole/control-implementation/lisp-container")

6.5 OLE automation

These examples illustrate using OLE automation:

(example-edit-file "com/automation/internet-explorer/simple")

(example-edit-file "com/automation/excel/pie-chart")

This is a simple example of an Automation collection interface. Follow the instructions in defsys.lisp:

(example-edit-file "com/automation/collections/defsys")

(example-edit-file "com/automation/collections/collection-test.idl")

(example-edit-file "com/automation/collections/client")

(example-edit-file "com/automation/collections/server")

(example-edit-file "com/automation/collections/compile-tlb")

This is an example of building and testing a CAPI application that can be controlled by Automation. Start with readme.txt:

(example-edit-file "com/automation/capi-application/readme.txt")

(example-edit-file "com/automation/capi-application/build")

(example-edit-file "com/automation/capi-application/defsys")

(example-edit-file "com/automation/capi-application/listapp.idl")

(example-edit-file "com/automation/capi-application/listapp.tlb")
6 Self-contained examples

These two files illustrate use of the CrystalDesignRunTime component:

(example-edit-file "com/automation/crystal-reports/deliver")
(example-edit-file "com/automation/crystal-reports/aubrowse")

This example illustrates using events with Internet Explorer:

(example-edit-file "com/automation/events/ie-events")

This is an example of building an Automation server without a GUI.

(example-edit-file "com/automation/cl-smtp/clsmtp-impl-build")
(example-edit-file "com/automation/cl-smtp/clsmtp.idl")
(example-edit-file "com/automation/cl-smtp/clsmtp.tlb")
(example-edit-file "com/automation/cl-smtp/clsmtp-impl")
(example-edit-file "com/automation/cl-smtp/server")
(example-edit-file "com/automation/cl-smtp/clsmtp-test")
Index

A
accessors
  interface-ref 54
  lisp-variant-type lisp-variant 114
  lisp-variant-value lisp-variant 114
  simple-i-dispatch-invoke-callback simple-i-dispatch 125
ActiveX controls  preface 8
add-ref function 27 1.6: Reference counting 12
automation-server-command-line-action function 28
automation-server-main function 28
automation-server-top-loop function 30

C
call-com-interface macro 31 1.8: Calling COM interface methods 12
call-com-object macro 32 1.9.2: The lifecycle of a COM object 19, 1.10: Calling COM object methods from Lisp 24
call-dispatch-get-property macro 87
call-dispatch-method macro 88
call-dispatch-put-property macro 89
Calling
  Automation methods: using a type library 3.3.1: Calling Automation methods using a type library 81
  Automation methods: without using a type library 3.3.2: Calling Automation methods without a type library 81
  COM interface methods 1.8: Calling COM interface methods 12
  COM object methods 1.10: Calling COM object methods from Lisp 24
check-result macro 34
classes
  com-object 38
  simple-i-dispatch 125
  standard-automation-collection 127
  standard-i-connection-point-container 128
  standard-i-dispatch 129
  standard-i-unknown 71
class factories 1.9.3: Class factories 20
class options
  :coclasse define-automation-component 97
  :coclasse-reusable-p define-automation-component 97
  :dont-implement define-com-implementation 47
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>```</td>
</tr>
<tr>
<td>:extra-interfaces define-automation-component 97</td>
</tr>
<tr>
<td>:inherit-from define-com-implementation 46</td>
</tr>
<tr>
<td>:interface define-automation-collection 96</td>
</tr>
<tr>
<td>:interfaces define-com-implementation 46, define-automation-component 97</td>
</tr>
<tr>
<td>:item-method define-automation-collection 96</td>
</tr>
<tr>
<td>:source-interfaces define-automation-component 97</td>
</tr>
<tr>
<td>:coclass class option define-automation-component 97</td>
</tr>
<tr>
<td>:coclass-reusable-p class option define-automation-component 97</td>
</tr>
<tr>
<td>co-create-guid function 34</td>
</tr>
<tr>
<td>co-initialize function 35 1.9.1: Steps required to implement COM interfaces 18</td>
</tr>
<tr>
<td>collections</td>
</tr>
<tr>
<td>implementing 3.4.3: Implementing collections 85</td>
</tr>
<tr>
<td>using 3.3.4: Using collections 83</td>
</tr>
<tr>
<td>com-dispatch-invoker-exception-error condition class 90</td>
</tr>
<tr>
<td>com-dispatch-invoker-exception-error-info function 91</td>
</tr>
<tr>
<td>com-error condition class 36</td>
</tr>
<tr>
<td>com-error-function-name function com-error 36</td>
</tr>
<tr>
<td>com-error-hresult function com-error 36</td>
</tr>
<tr>
<td>com-interface system class 37 1.8: Calling COM interface methods 12, 3.3.1: Calling Automation methods using a type library 81, 3.3.2: Calling Automation methods without a type library 82</td>
</tr>
<tr>
<td>com-interface-refguid function 38</td>
</tr>
<tr>
<td>COM interface types</td>
</tr>
<tr>
<td>i-dispatch 108</td>
</tr>
<tr>
<td>i-unknown 55</td>
</tr>
<tr>
<td>com-object class 38</td>
</tr>
<tr>
<td>com-object-destructor generic function 39 1.9.2: The lifecycle of a COM object 19</td>
</tr>
<tr>
<td>com-object-dispinterface-invoker generic function 92</td>
</tr>
<tr>
<td>com-object-from-pointer function 40</td>
</tr>
<tr>
<td>com-object-initialize generic function 41 1.9.2: The lifecycle of a COM object 19</td>
</tr>
<tr>
<td>com-object-query-interface generic function 41</td>
</tr>
<tr>
<td>compiling IDL files 1.1: Prerequisites 10</td>
</tr>
<tr>
<td>condition classes</td>
</tr>
<tr>
<td>com-dispatch-invoker-exception-error 90</td>
</tr>
<tr>
<td>com-error 36</td>
</tr>
<tr>
<td>connection points</td>
</tr>
<tr>
<td>implementing 3.4.4: Implementing connection points 85</td>
</tr>
<tr>
<td>using 3.3.5: Using connection points 83</td>
</tr>
<tr>
<td>CoTaskMemAlloc co-task-mem-alloc 43</td>
</tr>
<tr>
<td>co-task-mem-alloc function 42 1.8.1.3: In-out parameters 16</td>
</tr>
<tr>
<td>CoTaskMemFree co-task-mem-free 44</td>
</tr>
</tbody>
</table>
Index

co-task-mem-free function 43  1.8.1.2: Out parameters 14,  1.8.1.3: In-out parameters 16
co-uninitialize function 44
:count-function initarg standard-automation-collection 127
create-instance function 45
create-instance-with-events function 93
create-object function 94

D
: data-function initarg standard-automation-collection 127
define-automation-collection macro 95
define-automation-component macro 96
define-com-implementation macro 46  1.9.1: Steps required to implement COM interfaces 18
define-com-method macro 48  1.9.1: Steps required to implement COM interfaces 18
define-disinterface-method macro 98
defsystem member types
  :midl-file 60  1.2.2: Generating FLI definitions from COM definitions 10
  :midl-type-library-file 116  3.1.2: Generating FLI definitions from COM definitions 80
deliver function 1.2.4: Making a COM DLL with LispWorks 11, automation-server-main 30, automation-server-top
  -loop 31
destruction 1.9.2: The lifecycle of a COM object 19
discard-connection function do-connections 102
disconnect-standard-sink function 100
disinterface 3.1.3: Reducing the size of the converted library 80,  3.4.1: A complete implementation of an Automation server 84,  3.4.2: A simple implementation of a single Automation interface 84,  com-object-disinterface-invoke 92, define-
  disinterface-method 98
:dll-exports delivery keyword 1.2.4: Making a COM DLL with LispWorks 11, set-register-server-error-reporter 69
do-collection-items macro 100
  do-connections macro 101
: dont-define class option define-com-implementation 47
dual interface 3.1.3: Reducing the size of the converted library 80,  3.4.1: A complete implementation of an Automation server 84

E
editor commands
  Function Arglist 5.4.2: Argument lists 136
  Insert GUID 5.4.1: Inserting GUIDs 136
environment variables
  INCLUDE midl 58, midl-set-import-paths 60
errors
  handling in Automation 3.3.6: Error handling 83
  handling in COM 1.8.2: Error handling 17
  reporting 3.4.5: Reporting errors 85
events
  see connection-points 3.4.4: Implementing connection points 85
Index

:extra-interfaces class option define-automation-component 97

F

find-clsid function 49
find-component-tlb function 102
find-component-value function 103

FLI type descriptors

hrresult 53
refguid 63
refiid 64

Function Arglist editor command 5.4.2: Argument lists 136

:function-name initarg com-error 36

functions

add-ref 27

automation-server-command-line-action 28
automation-server-main 28
automation-server-top-loop 30
co-create-guid 34
co-initialize 35 1.9.1: Steps required to implement COM interfaces 18
com-dispatch-invokes-exception-error-info 91
com-error-function-name com-error 36
com-error-hresult com-error 36
com-interface-refguid 38
com-object-from-pointer 40
co-task-mem-alloc 42 1.8.1.3: In-out parameters 16
co-task-mem-free 43 1.8.1.2: Out parameters 14, 1.8.1.3: In-out parameters 16
co-uninitialize 44
create-instance 45
create-instance-with-events 93
create-object 94
discard-connection do-connections 102
disconnect-standard-sink 100
find-clsid 49
find-component-tlb 102
find-component-value 103
get-active-object 105
get-error-info 106
get-i-dispatch-name 107
get-i-dispatch-source-names 108
get-object 50
guid-equal 51
guid-to-string 52

143
Index

hresult-equal 53
interface-connect 109
interface-disconnect 110
invoke-dispatch-get-property 111
invoke-dispatch-method 112
invoke-dispatch-put-property 113
make-factory-entry 55 1.9.1: Steps required to implement COM interfaces 18, 1.9.3: Class factories 20
make-gid-from-string 56
make-lisp-variant 115
midl 57 1.2.2: Generating FLI definitions from COM definitions 10
midl-default-import-paths 59
midl-set-import-paths 60
print-i-dispatch-methods 117
query-interface 61
query-simple-i-dispatch-interface 118
refguid-interface-name 63
register-active-object 119
register-class-factory-entry 65 1.9.1: Steps required to implement COM interfaces 18, 1.9.3: Class factories 20
register-server 66
release 67
revoke-active-object 120
server-can-exit-p 67
server-in-use-p 67
set-automation-server-exit-delay 68
set-error-info 120
set-i-dispatch-event-handler 121
set-register-server-error-reporter 69
set-variant 123
simple-i-dispatch-interface-name simple-i-dispatch 125
simple-i-dispatch-refguid simple-i-dispatch 125
start-factories 72 1.9.1: Steps required to implement COM interfaces 18, 1.9.3: Class factories 20
stop-factories 72
unregister-server 74

G
Garbage collection 1.9.2: The lifecycle of a COM object 19
generic functions
com-object-destructor 39 1.9.2: The lifecycle of a COM object 19
com-object-dispinterface-involve 92
com-object-initialize 41 1.9.2: The lifecycle of a COM object 19
com-object-query-interface 41
simple-i-dispatch-callback-object 126
Index

get-active-object function 105
get-error-info function 106  1.8.2: Error handling 17
get-i-dispatch-name function 107
get-i-dispatch-source-names function 108
get-object function 50
guid-equal function 51
guid-to-string function 52

H
hresult FLI type descriptor 53
:hresult initarg com-error 36
hresult-equal function 53

I
i-dispatch COM interface type 108

IDL
compiling  1.1: Prerequisites 10
iid_is attribute  1.8.1.2: Out parameters 15

INCLUDE environment variable  58, midl 58, midl-set-import-paths 60
inheritance  1.9.5: Inheritance 20
 :inherit-from class option define-com-implementation 46

initialization
CLOS object  1.9.2: The lifecycle of a COM object 19
COM object  1.9.2: The lifecycle of a COM object 19

in-out parameters  1.8.1.3: In-out parameters 16, 1.9.6.4: In-out parameters 24, 1.10.1.3: In-out parameters 25, 3.3.3: Data conversion when calling Automation methods 82

in parameters  1.8.1.1: In parameters 13, 1.9.6.2: In parameters 23, 1.10.1.1: In parameters 25, 3.3.3: Data conversion when calling Automation methods 82

Insert GUID editor command  5.4.1: Inserting GUIDs 136
:interface class option define-automation-collection 96

interface-connect function 109
interface-disconnect function 110

:interface-name initarg simple-i-dispatch 125
interface-ref accessor 54

:interfaces class option define-com-implementation 46, define-automation-component 97
:invoke-callback initarg simple-i-dispatch 125

invoke-dispatch-get-property function 111
invoke-dispatch-method function 112

invoke-dispatch-put-property function 113

:item-generator-function initarg standard-automation-collection 127
item-lookup-function initarg standard-automation-collection 127

item-method class option define-automation-collection 96
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>:items-function initarg standard-automation-collection 127</td>
</tr>
<tr>
<td>i-unknown COM interface type 55</td>
</tr>
<tr>
<td>L</td>
</tr>
<tr>
<td>lisp-variant system class 114</td>
</tr>
<tr>
<td>lisp-variant-type accessor lisp-variant 114</td>
</tr>
<tr>
<td>lisp-variant-value accessor lisp-variant 114</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>macros</td>
</tr>
<tr>
<td>call-com-interface 31</td>
</tr>
<tr>
<td>call-com-object 32 1.9.2: The lifecycle of a COM object 19</td>
</tr>
<tr>
<td>call-dispatch-get-property 87</td>
</tr>
<tr>
<td>call-dispatch-method 88</td>
</tr>
<tr>
<td>call-dispatch-put-property 89</td>
</tr>
<tr>
<td>check-hresult 34</td>
</tr>
<tr>
<td>define-automation-collection 95</td>
</tr>
<tr>
<td>define-automation-component 96</td>
</tr>
<tr>
<td>define-com-implementation 46 1.9.1: Steps required to implement COM interfaces 18</td>
</tr>
<tr>
<td>define-com-method 48 1.9.1: Steps required to implement COM interfaces 18</td>
</tr>
<tr>
<td>define-dispinterface-method 98</td>
</tr>
<tr>
<td>do-collection-items 100</td>
</tr>
<tr>
<td>do-connections 101</td>
</tr>
<tr>
<td>query-object-interface 62 1.9.2: The lifecycle of a COM object 19</td>
</tr>
<tr>
<td>s_ok 70</td>
</tr>
<tr>
<td>succeeded 73</td>
</tr>
<tr>
<td>with-coclass 130</td>
</tr>
<tr>
<td>with-com-interface 75</td>
</tr>
<tr>
<td>with-com-object 76</td>
</tr>
<tr>
<td>with-dispatch-interface 131</td>
</tr>
<tr>
<td>with-query-interface 77</td>
</tr>
<tr>
<td>with-temp-interface 78</td>
</tr>
<tr>
<td>make-factory-entry function 55 1.9.1: Steps required to implement COM interfaces 18, 1.9.3: Class factories 20</td>
</tr>
<tr>
<td>make-guid-from-string function 56</td>
</tr>
<tr>
<td>make-lisp-variant function 115</td>
</tr>
<tr>
<td>making a COM DLL 1.2.4: Making a COM DLL with LispWorks 11</td>
</tr>
<tr>
<td>midl function 57 1.2.2: Generating FLI definitions from COM definitions 10</td>
</tr>
<tr>
<td>midl-default-import-paths function 59</td>
</tr>
<tr>
<td>midl.exe 1.2.2: Generating FLI definitions from COM definitions 10, 1.8.1: Data conversion when calling COM methods 13, 1.9.6: Data conversion in define-com-method 22</td>
</tr>
<tr>
<td>:midl-file defsystem member type 60 1.2.2: Generating FLI definitions from COM definitions 10</td>
</tr>
<tr>
<td>midl-set-import-paths function 60</td>
</tr>
</tbody>
</table>
Index

query-object-interface macro 62 1.9.2: The lifecycle of a COM object 19
query-simple-i-dispatch-interface function 118
:quit-when-no-windows delivery keyword automation-server-top-loop 31

R
refguid FLI type descriptor 63
refguid-interface-name function 63
reffi FLI type descriptor 64 1.7: Querying for other COM interface pointers 12
register-active-object function 119
register-class-factory-entry function 65 1.9.1: Steps required to implement COM interfaces 18, 1.9.3: Class factories 20
register-server function 66
registry
component values find-component-value 103
guid find-clsid 49
ProgID find-clsid 49
type library versions find-component-tlb 102
release function 67 1.6: Reference counting 12
retval attribute 3.3.3: Data conversion when calling Automation methods 82
revoke-active-object function 120

S
save-image function 1.2.4: Making a COM DLL with LispWorks 11, automation-server-main 30
Self-contained examples
ActiveX controls 6.4: Building an ActiveX control 138
aggregation 6.2: Aggregation 137
argument passing 6.1: Argument passing 137
Automation 3.5: Examples of using Automation 86
calling and implementing COM methods 6.1: Argument passing 137
COM/Automation 6: Self-contained examples 137, 6.4: Building an ActiveX control 138
Controlling an Automation application 3.5: Examples of using Automation 86
embedding external components 6.3: OLE embedding of external components 137
event handlers 6.3: OLE automation 139
events 6.5: OLE automation 139
Getting events from COM interfaces 3.5: Examples of using Automation 86
OLE automation 6.5: OLE automation 138
OLE embedding 6.3: OLE embedding of external components 137
server-can-exit-p function 67
server-in-use-p function 67
set-automation-server-exit-delay function 68
set-error-info function 120 define-com-method 49, 3.4.5: Reporting errors 85
set-i-dispatch-event-handler function 121
set-register-server-error-reporter function 69
Index

set-variant function 123

simple-i-dispatch class 125

simple-i-dispatch-callback-object generic function 126

simple-i-dispatch-interface-name function simple-i-dispatch 125

simple-i-dispatch-invoke-callback accessor simple-i-dispatch 125

simple-i-dispatch-refguid function simple-i-dispatch 125

size_is attribute 1.8.1.1: In parameters 13, 1.8.1.2: Out parameters 15, 1.8.1.3: In-out parameters 16, 1.9.6.2: In parameters 23, 1.9.6.3: Out parameters 23, 1.9.6.4: In-out parameters 24, 1.10.1.1: In parameters 25, 1.10.1.2: Out parameters 25

s_ok macro 70

source attribute define-automation-component 97

source interfaces 3.4.4: Implementing connection points 85

:source-interfaces class option define-automation-component 97

standard-automation-collection class 127

standard-i-connection-point-container class 128

standard-i-dispatch class 129

standard-i-unknown class 71

start-factories function 72 1.9.1: Steps required to implement COM interfaces 18, 1.9.3: Class factories 20

stop-factories function 72

string attribute 1.8.1.1: In parameters 13, 1.8.1.2: Out parameters 14, 1.8.1.3: In-out parameters 16, 1.9.6.2: In parameters 23, 1.9.6.3: Out parameters 23, 1.9.6.4: In-out parameters 24, 1.10.1.1: In parameters 25, 1.10.1.3: In-out parameters 25

succeeded macro 73

system classes

com-interface 37 1.8: Calling COM interface methods 12, 3.3.1: Calling Automation methods using a type library 81, 3.3.2: Calling Automation methods without a type library 82

lisp-variant 114

T

tools

COM Implementation Browser 5.1: The COM Implementation Browser 133

COM Interface Browser 5.3: The COM Interface Browser 135

COM Object Browser 5.2: The COM Object Browser 135

type libraries 3.1.2: Generating FLI definitions from COM definitions 80

U

unimplemented methods 1.9.4: Unimplemented methods 20

unregister-server function 74

V

vararg attribute 1.9.6.1: FLI types 23, 3.3.3: Data conversion when calling Automation methods 82, define-dispinterface-method 99
### Index

**W**

Windows registry  
| find-clsid 49 | find-component-tlb 102 | find-component-value 103 |

**with-coclass** macro  130

**with-com-interface** macro  75  
1.8 : Calling COM interface methods 12

**with-com-object** macro  76  
1.10 : Calling COM object methods from Lisp 24

**with-dispatch-interface** macro  131

**with-query-interface** macro  77  
1.7 : Querying for other COM interface pointers 12

**with-temp-interface** macro  78  
1.6 : Reference counting 12