Harlequin Dylan

OLE, COM, ActiveX and DBMS Reference

Library Reference

Version 2.0 Beta

Copyright and Trademarks

Harlequin Dylan: Library Reference: OLE, COM, ActiveX, and DBMS

Version 2.0 Beta

May 1999

Part number: 3DPDTZ-15MG

http://www.harlequin.com/

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
UK

telephone +44 1223 873 800
fax +44 1223 873 873

North America:

Harlequin Incorporated
One Cambridge Center
Cambridge, MA 02142
USA

telephone +1 617 374 2400
fax +1 617 252 6505

Asia Pacific:

Harlequin Australia Pty. Ltd.
Level 12
12 Moore Street
Canberra, ACT 2601
Australia

telephone +61 2 6206 5522
fax +61 2 6206 5525

Contents

1

The SQL-ODBC Library

This chapter discusses Harlequin Dylan's database interoperability support. It assumes a basic knowledge of SQL and Microsoft's Open Database Connectivity™ (ODBC) interface.

1.1 Introduction

Harlequin Dylan's SQL-ODBC library provides a generic Dylan protocol for interfacing applications to any database management system (DBMS) supporting Microsoft's Open Database Connectivity™ (ODBC) interface and the industry-standard database query language, SQL. The SQL-OBDC library supports the full SQL language defined in the ANSI SQL-89 and ANSI SQL-92 specifications, as well as any extensions defined by a DBMS.

A low-level interface to the Microsoft ODBC API is also available in the ODBC-FFI library. Harlequin built the ODBC-FFI library using the C-FFI library and the same C-to-Dylan name mapping scheme as described in the Win32 API FFI library documentation. See the *C FFI and Win32* library reference for details of that scheme. The ODBC-FFI library is otherwise undocumented.

1.1.1 Implementation

The SQL-ODBC library is built on top of a generic SQL library. This SQL library does not include the low-level code necessary to communicate with any particular DBMS. In itself, the SQL library simply provides a convenient high-level mechanism for integrating database operations into Dylan applications. It is designed to form the high-level part of "implementation libraries" that contain lower-level code to supporting a particular DBMS protocol, such as ODBC. The SQL-ODBC library is, then, one such implementation library.

Our intention is that the SQL library will provide a common high-level Dylan interface to any DBMS. Applications written using the SQL-ODBC library will therefore be simple to port to any future DBMSes for which implementation libraries are written.

1.1.2 Using the SQL-ODBC library in applications

The SQL-ODBC library is available to applications as the **`SQL-ODBC`** library, which exports the modules **`SQL-ODBC`** and **`SQL`**. (You should not need to use the **`SQL`** module, but it will be visible during debugging sessions.)

1.1.3 SQL Standards

The SQL-ODBC library supports the full SQL language specified in the ANSI specification X3.135-1992, *Database Language — SQL*, commonly known as SQL-92, as well as any DBMS extensions.

1.1.4 Object-oriented languages and relational databases

The SQL-ODBC library does not provide the means to "objectify" a relational database or an SQL statement. That is, it does not treat a databases or statements in an object-oriented fashion or provide support for doing so.

This is because the object-oriented programming model is very different from the relational database model. The two most significant differences follow.

First, the relational database model has only simple datatypes (string, integer, floating point number, and so on) and does not provide a means of defining new types, as object-oriented programming languages do.

Second, objects in an object-oriented program have unique identities that allow two objects of the same value to be distinguished from one another. By contrast, SQL rows (the SQL notion nearest to the notion of an object) do not have unique identities: if two rows in a given table have identical values, they are indistinguishable.

1.1.5 Result-retrieval protocol

The SQL-ODBC library provides an abstract Dylan protocol for handling SQL *result sets*, the means by which SQL **SELECT** statement results are retrieved. The library allows result sets to be processed as Dylan collections. The various Dylan collection protocols and functions work as you would expect on a result set.

1.1.6 Processing results

SQL **SELECT** statements return database records. You process the results of an SQL **SELECT** statement using a result set. *Result sets* are the focal point of the SQL-ODBC library's encapsulation of the protocol for retrieving database records. Using result sets allows you to concentrate on the logic of your application instead of the logic of record retrieval.

Result sets retrieve their records from the database synchronously. As result sets retrieve their records, you can specify conversion of records to application-specific objects which are added to the result set in place of the record. Result sets retrieve their records one at a time.

1.1.7 Bridging the object-relational gap

Relational DBMSes do not in general deal with objects or classes. Since Dylan is an object-oriented language, this creates a gap between Dylan and the DBMS.

The SQL-ODBC library bridges this gap by allowing you to specify a liaison function for results. A *liaison* function acts as an interpreter for results, taking the records retrieved from the relational DBMS and converting each into suitable Dylan objects. A default liaison method exists for use in situations where your application does not know the appropriate conversion, for example when processing SQL **SELECT** statements typed in by the application user. The default method transforms each record retrieved into a Dylan collection, where each element of the collection corresponds to a column of the record. See Section 1.5.4 on page 33 for more on liaison functions.

1.1.8 Error handling

As in any application, errors at run time can occur when applications talk to databases. The SQL-ODBC library captures the errors and warnings that a DBMS generates and signals a corresponding Dylan error or warning condition. Your application can then process the condition using the Dylan condition system.

1.1.9 Examples used in this document

The following tables depict example database tables to which this document's code examples refer.

Title	Publisher	ISBN
An Introduction to Database Systems	Addison Wesley	0-201-14201-5
Transaction Processing: Concepts and Techniques	Morgan Kaufmann	1-55860-190-2
Fundamentals of Database Systems	Benjamin/Cummings	0-8053-1748-1
Relational Database Writings, 1991-1994	Addison-Wesley	0-201-82459-0

Table 1.1 Table "Book" used in this document's code examples.

Author ID	Last Name	First Name
1	Date	Chris
2	Gray	Jim

Table 1.2 Table "Author" used in this document's code examples.

Author ID	Last Name	First Name
3	Reuter	Andreas
4	Elmasri	Ramez
5	Navathe	Shamkant

Table 1.2 Table “Author” used in this document’s code examples.

Author_ID	ISBN
1	0-201-14201-5
2	1-55860-190-2
3	1-55860-190-2
4	0-8053-1748-1
5	0-8053-1748-1
1	0-201-82459-0

Table 1.3 Table “Book_author” used in this document’s code examples.

1.2 Connecting to a database

Before it can query a database, your application must connect to it. Most DBMSes operate a form of login procedure to verify connections, using a user name and password for the purpose. The popular DBMSes each have different protocols for identifying themselves, their users, their databases, and connections to those databases.

The SQL-ODBC library provides a general-purpose connection protocol that is not specific to any DBMS, and represents DBMSes, databases, database connections, user names and passwords with generic Dylan classes, thereby hiding the idiosyncrasies of the various DBMSes from Dylan applications. The classes that the SQL-ODBC library defines are shown in Table 1.4.

Entity	Abstract Dylan class	SQL-ODBC class
DBMS	<dbms>	<odbc-dbms>
Database	<database>	<odbc-database>
User name and password	<user>	<odbc-user>
Active connection	<connection>	<odbc-connection>

Table 1.4 Dylan DBMS classes.

You should create DBMS-specific instances of these classes to connect to a database.

1.2.1 The <DBMS> class

The **`<dbms>`** class identifies a database management system (DBMS) to a client application. Implementation libraries like SQL-ODBC supply an instantiable subclass of **`<dbms>`** to provide whatever implementation is necessary for identifying a DBMS to an application.

<dbms> *Abstract class*

Description

Instances of this class identify a particular database management system (DBMS) to an application. It is the root class of all DBMS classes, and a subclass of **`<object>`**.

The SQL-ODBC library defines an instantiable class **`<odbc-dbms>`** for identifying an ODBC DBMS.

1.2.2 The <USER> class

The **`<user>`** class identifies a user to a DBMS. Exactly what a “user” means depends on the DBMS. Implementation libraries like SQL-ODBC supply an instantiable subclass of **`<user>`** to provide whatever implementation is necessary for identifying a user to a specific DBMS.

When connecting to a DBMS that did not have any users per se, instances of **<user>** would merely satisfy the API protocol, and would not identify a specific user — any instance of **<user>** would identify all users to the DBMS. However, most DBMSes do require a user name and password to identify a specific user. Indeed, some DBMSes require stringent authorization information in order to identify a user, such as multiple passwords.

<user> *Abstract class*

Description

Instances of this class identify a user to a DBMS. It is the root class of all user classes, and a subclass of **<object>**.

The **user:** init-keyword takes an instance of **<string>** that should be a valid user name for the DBMS in question. The **password:** init-keyword should be the password that accompanies the user name.

If you apply **make** to the class **<user>** within the scope of the **with-dbms** macro, an instance of a DBMS-specific user class is created.

The SQL-ODBC library defines a class **<odbc-user>** for identifying a user to an ODBC DBMS.

1.2.3 The <DATABASE> class

The **<database>** class identifies a database to a DBMS. Exactly what a database is depends on the DBMS in use. Implementation libraries like SQL-ODBC supply an instantiable subclass of **<database>** to provide whatever implementation is necessary for identifying a database to a specific DBMS.

<database> *Abstract class*

Description

Instances of this class identify a database to a DBMS. It is the root class of all database classes, and a subclass of **<object>**.

If you apply **`make`** to the class **`<database>`** within the scope of the **`with-dbms`** macro, an instance of a DBMS-specific database class is created.

The SQL-ODBC library defines a class **`<odbc-database>`** for identifying a database to an ODBC DBMS.

1.2.4 The <CONNECTION> class

The **`<connection>`** class represents a database connection. More formally, we can say that it identifies a context in which a client application can execute SQL statements. The exact composition of a connection depends on the DBMS and the client platform. Implementation libraries like SQL-ODBC define a subclass of **`<connection>`** that implements the necessary requirements to identify the execution context to the client application.

<connection> *Abstract class*

Description Instances of this class identify an execution context to a client application. It is the root class of all connection classes, and a subclass of **`<object>`**.

The SQL-ODBC library defines a class **`<odbc-connection>`**, and instances of this class are created by the SQL-ODBC library when a connection is made to a database. A client application should not create instances of the **`<connection>`** class or any of its subclasses.

1.2.5 Connection protocol functions, methods, and macros

with-dbms *Statement macro*

Summary Considers *dbms* to be the DBMS in use throughout *body*.

Signature

```
with-dbms(dbms)
  body
end [with-dbms]
```

Arguments *dbms* An instance of **<dbms>**.

Description Considers *dbms* to be the DBMS in use throughout *body*. If you apply **make** to any of the classes **<user>**, **<database>**, or **<sql-statement>** in *body*, this macro creates an instance of the subclass suitable for *dbms*. This means you can write code in *body* that does not refer to a specific DBMS type, making it easier to write inter-DBMS applications. For example, you can write this:

```
let dbms = make(<odbc-dbms>);
with-dbms(dbms)
  let user = make(<user>,
   name: "foo", password: "bar");
  let db = make(<database>,
   datasource-name: "db.world");
  let connection = connect(db, user);
  …
end with-dbms;
```

Instead of writing this:

```
let dbms = make(<odbc-dbms>);
let user = make(<odbc-user>,
  name: "foo".
  password: "bar");
let db = make (<odbc-database>, datasource-name:
"db.world");
let connection = connect(db, user, dbms: dbms);
…
```

dbms *Generic function*

Summary Returns the DBMS object associated with the *connection*.

Signature **dbms** ***connection*** **=>** ***dbms***

Arguments *connection* An instance of **<connection>**.

Values *dbms* An instance of **<dbms>**.

Description — Returns the DBMS object associated with the *connection*.

database — *Generic function*

Summary — Returns the database object associated with the connection.

Signature — **`database`** ***connection*** **`=>`** ***database***

Arguments — *connection* — An instance of **`<database>`**.

Values — *database* — An instance of **`<database>`**.

Description — Returns the database object associated with *connection*.

user — *Generic function*

Summary — Returns the user object associated with the connection.

Signature — **`user`** ***connection*** **`=>`** ***usere***

Arguments — *connection* — An instance of **`<connection>`**.

Values — *user* — An instance of **`<user>`**.

Description — Returns the user object associated with *connection*.

1.2.6 Connecting and disconnecting

The SQL-ODBC library provides DBMS-independent functions to connect to and disconnect from databases. Connecting to a database establishes a context (an instance of **`<connection>`**) in which SQL statements may be executed within an application. You can make connections by calling the **`connect`** function on a DBMS-specific instance of **`<database>`** and **`<user>`**.

An application can connect to multiple databases served by a DBMS if the DBMS supports the feature. Multiple-connection support can be determined by calling the **`multiple-connections?`** function on the DBMS object.

Keeping connections open requires system resources. An application can disconnect from connections that it no longer needs in order to reduce its use of system resources. When the application terminates, the SQL-ODBC library disconnects all open connections. If a connection is not explicitly terminated using the **`disconnect`** generic function, and a client application has no references to it, the connection is terminated when the garbage collector notices that the object can be reclaimed. After a connection has been disconnected, the **`<connection>`** object cannot be reused, and so references to it should be dropped.

connect *Generic function*

Summary: Connects to the specified database as the specified user.

Signature: **`connect`** ***database user*** **`=>`** ***connection***

Arguments:

Arguments	*database*	An instance of **`<database>`**.
	user	An instance of **`<user>`**.

Values:

Values	*connection*	An instance of **`<connection>`**.

Description: Connects to the database *database* as the user *user*. Returns an instance of **`<connection>`** representing the connection to *database*.

connections *Generic function*

Summary: Returns a sequence of all active connections against the specified DBMS.

Signature: **`connections #key`** ***dbms*** **`=>`** ***connection-sequence***

Arguments *dbms* An instance of **`false-or(<dbms>)`**.

Values *connection-sequence*
An instance of **`<sequence>`**.

Description Returns a sequence of all active connections against the DBMS-specified by *dbms*. Returns a sequence of all active connections if *dbms* is **`#f`**.

default-connection *Generic function*

Summary Returns the connection established by the **`with-database`** or **`with-connections`** macro.

Signature **`default-connection () =>`** ***connection***

Arguments None.

Values *connection* An instance of **`<connection>`**.

Description Returns the *connection* established by the **`with-database`** or **`with-connection`** macro. Signals the condition **`<connection-not-specified>`** if a connection has not been established or if this function is invoked outside of the scope of a **`with-database`** or **`with-connection`** macro.

disconnect *Generic function*

Summary Terminates a connection.

Signature **`disconnect`** ***connection*** **`#key`** ***terminate-statements*** **`=> ()`**

Arguments *connection* An instance of **`<connection>`**.

Values *terminate-statements*
An instance of **`<boolean>`**. Default value: **`#f`**.

Description Terminates *connection*. If any SQL statements are executing asynchronously, or an SQL **`SELECT`** statement has results which have not been retrieved, a condition is signaled, unless *terminate-statements* is **`#t`**.

disconnect-all *Generic function*

Summary Terminates all open connections served by the DBMS.

Signature **`disconnect-all #key`** ***dbms*** **`=> ()`**

Arguments *dbms* An instance of **`false-or(<symbol>)`**. Default value: **`#f`**.

Description Terminates all open connections served by *dbms*. If *dbms* is **`#f`** (the default value), all connections on all DBMSs are terminated.

with-connection *Statement macro*

Summary Within the dynamic scope of the body, establishes the default connection.

Signature

```
with-connection(connection)
  body
end [with-connection]
```

Arguments *connection* An instance of **`<connection>`**.

Description Within the dynamic scope of *body*, establishes the default connection as *connection*. The value of *connection* must be an instance of **<connection>** returned from the method **connect**. The result of this macro is the last expression executed within *body*.

Example:

This example queries the ODBC database library for the number of books published by Addison-Wesley and the number of books published by McGraw-Hill. This example is written in an unorthodox manner (two connections to the same database) to show that SQL statements may be executed against different established connections using the **with-connection** macro. The **<result-set>** class is discussed on page 32.

```
begin
  let db = make (<odbc-database>, name: "library");
  let user = make(<odbc-user>, user: "andrew",
                  password: "foobar");
  let a-connection :: <connection> = connect(db, user);
  let b-connection :: <connection> = connect(db, user);

  let aw-query = make (<odbc-sql-statement>,
    text: "select count(*)
    from book
    where publisher = "Addison-Wesley"
  let addison-wesley-library :: <result-set> =
    with-connection(a-connection);
      executfe(aw-query),
    end with-connection;
  let first-record = addison-wesley-library[0];
  let addison-wesley-library-count = first-record[0];
  let mh-query = make (<odbc-sql-statement>, test:
        select count(*)
        from book
        where publisher = "McGraw-Hill"

  let mcgraw-hill-library :: <result-set> =
    with-connection(b-connection);
      execute(mh-query),
    end with-connection;
  let first-record = mcgraw-hill-library[0];
  let mcgraw-hill-library-count = first-record[0];
```

```
    disconnect-all();
    values(addison-wesley-library-count,
           mcgraw-hill-library-count);
  end;

  => 2
     0
```

with-database *Statement macro*

Summary

Executes all SQL statements appearing in the body of the code against the database for the user.

Signature

with-database(*database*, *user*)
  body
end [with-database]

Arguments

database An instance of **<database>**.

user An instance of **<user>**.

Description

Executes all SQL statements appearing in *body* against *database* for *user*.

This macro applies the **connect** function to its parameters, and establishes the resulting connection as the default connection within the dynamic scope of *body*. The connection is terminated when execution leaves the scope of *body*.

The result of this macro is the last expression executed in *body*.

This example queries the ODBC database **library** for a list of titles contained within the **book** table. The **with-database** macro establishes the **library** database as the database to execute the query against.

```
with-database(make(<odbc-database>, name: "library"),
              make(<odbc-user>, user: "andrew",
                   password: "foobar"));
  let query = make (<odbc-sql-statement>,
    text: "select title from book"),
    execute (query);
end with-database;
```

```
=> #(#("An Introduction to Database Systems",
     "Transaction Processing: Concepts and Techniques",
     "Fundamentals of Database Systems",
     "Relational Database Writings, 1991-1994"))
```

1.3 Executing SQL statements

The SQL-ODBC library provides a way of processing SQL statements: the **execute** function, which you must apply to instances of the **<sql-statement>** class.

1.3.1 The null value

SQL offers the *null value* to represent missing information, or information that is not applicable in a particular context. All columns of a table can accept the null value — unless prohibited by integrity constraints — regardless of the domain of the column. Hence, the null value is included in all domains of a relational database and can be viewed as an out-of-band value.

Relational database theory adopted a three-valued logic system — “true”, “false”, and “null” (or “unknown”) — in order to process expressions involving the null value. This system has interesting (and sometimes frustrating) consequences when evaluating arithmetic and comparison expressions. If an operand of an arithmetic expression is the null value, the expression evaluates to the null value. If a comparand of a comparison expression is the null value, the expression may evaluate to the null / unknown truth-value.

For example:

- a + b, where a contains the null value or b contains the null value, evaluates to the null value
- a + b, where a contains the null value and b contains the null value, evaluates to the null value

- a = b, where a contains the null value or b contains the null value, evaluates to unknown
- a = b, where a contains the null value and b contains the null value, evaluates to unknown
- a | b, where a is true and b contains the null value, evaluates to true
- a & b, where a is false and b contains the null value, evaluates to false

The SQL **`SELECT`** statements return records for which the **`WHERE`** clause (or **`WHERE`** predicate) evaluates to true (not to false and not to the null value). In order to test for the presence or absence of the null value, SQL provides a special predicate of the form

column-name **`is [not] null`**

The null value is effectively a universal value that is difficult to use efficiently in Dylan. To identify when null values are returned from or need to be sent to a DBMS server, the SQL-ODBC library supports indicator objects. *Indicator objects* indicate when a column of a record retrieved from a database contains the null value, or when a client application wishes to set a column to the null value.

<null-value> *Sealed concrete class*

Superclasses **`<object>`**

Description Instances of this class represent the canonical null value. This class is the root class for all null-value classes.

$null-value *Constant*

Description References the canonical null value. It is an instance of **`<null-value>`**.

1.3.2 Input indicators and output indicators

It is difficult for database applications written in traditional programming languages to represent the semantics of the null value, because it is a universal value which is in the domain of all types, and the three-valued logic system which accompanies null values does not easily translate to the two-value logic system in traditional programming languages.

In Dylan, a universal value can be achieved if we ignore type specialization, but this inhibits optimization and method dispatching. Even if we were to forgo type specialization, the evaluation of arithmetic and comparison expressions is a problem since Dylan's logic system is boolean and not three-valued. Therefore, the SQL-ODBC library has a goal of identifying null values and translating them into Dylan values that can be recognized as representing null values.

In order to identify null values during SQL statement processing, the **`<sql-statement>`** class supports an input indicator and output indicator. An *input indicator* is a marker value or values which identifies an input host variable as containing the null value. An *output indicator* is a substitution value which semantically identifies columns of a retrieved record as containing the null value.

If the SQL-ODBC library encounters a null value when retrieving records from a database, and there is no appropriate indicator object, it signals a **`<data-exception>`** condition. The condition is signaled from result-set functions (including the collection protocol) and not the **`execute`** function.

During the execution of an SQL statement to which an input indicator value was supplied, each input host variable is compared (with the function **`\==`**) to the input indicator and, if it holds the input indicator value, the null value is substituted for it.

The input indicator may be a single value or a sequence of values. A single value is useful when it is in the domain of all input host variables; if the host variables have not been specialized, any newly created value will do. Otherwise, a sequence of values must be used. Input indicators that are general instances of **`<sequence>`** use their positional occurrence within the SQL statement as the key for the sequence.

The SQL **SELECT** statement is the only SQL statement that returns non-status results back to the client application. During the retrieval of these results, the SQL-ODBC library substitutes the output indicator, if supplied, for null values found in the columns of each record.

The output indicator may be a single value or a sequence of values. If the output indicator is a general instance of **<sequence>**, the element of the sequence whose key corresponds to the column index is used as the substitution value. Otherwise, the output indicator value itself is used as the substitution value.

1.3.3 The SQL statement class

The **<sql-statement>** class represents SQL statements and their indicator values and coercion policy. You can use this class to represent any SQL statement, be it static or dynamic. You can send SQL statements to the DBMS for execution by calling the **execute** function on an instance of **<sql-statement>**. The **execute** function returns the results of executing the SQL statement, if there are any.

In the **make** method on **<sql-statement>**, you can specify that values should be substituted into the SQL statement when it is executed. You do not specify the values until calling **execute** on the statement, when you can pass the substitution values with the **parameter:** keyword.

The values are substituted wherever a question mark (?) occurs in the SQL statement string. We call the question marks *anonymous host variables* because there is no Dylan variable name. Substitution occurs positionally: the first value replaces the first anonymous host variable, the second value replaces the second anonymous host variable, and so on. If the number of values is greater than the number of anonymous host variables, the extra parameters are ignored. If the number of anonymous host variables is greater than the number of parameters, a condition is signaled.

When the SQL statement is **SELECT**, you can also specify a result-set policy and a liaison function in the call to **execute**. A *result-set policy* describes behavioral and performance characteristics of the result-set object that the **execute** function returns. A *liaison* function creates Dylan objects from the records retrieved from the database. These objects become the elements of the result set instead of the record object.

<database-statement> *Abstract class*

Superclasses **`<object>`**

Description This class represents statements which can be executed by a DBMS server.

execute *Open generic function*

Summary Sends a database-statement to the DBMS server to be executed and collects its return values.

Signature **`execute`** ***database-statement*** **`#key`** ***all-keys*** **`=>`** ***result-set***

Arguments

database-statement
An instance of **`<string>`**.

result-set An instance of **`<result-set>`**.

Description Allows a string to create and send a *database-statement* to the DBMS server to be executed, and returns *result-set*. If the statement does not cause the DBMS to return any values, this generic function's return value will be an instance of **`<empty-result-set>`** (a subclass of **`<result-set>`**). See also, "execute" on page 23.

<sql-statement> *Abstract class*

Superclasses **`<database-statement>`**

Description Instances of this class represent an SQL statement and its state. This class has the following init-keywords:

`text:` An instance of **`<string>`**. Required. Contains the text of the SQL statement. If you want to include host variables, place a question mark (**?**) at the point in the string at which you want a host variable to be substituted.

output-indicator:
An instance of **<object>**. The output indicator is a substitution value to be used whenever the column of a retrieved record contains the null value.

input-indicator:
An instance of **<object>**. The input indicator is a marker value used to identify null values in host variables.

coercion-policy:
An instance of **false-or(<coercion-policy>)**. The coercion policy is a sequence of functions, or the value **$default-coercion**, or the value **$no-coercion**, used to perform data coercion when the SQL statement to be executed is a **SELECT** statement.

datatype-hints:
An instance of **false-or(<sequence>)**. This is a hint for parameter binding when the SQL statement to be executed is a **SELECT** statement.

coercion-policy *Method*

Summary Returns the coercion policy for an SQL statement.

Signature **coercion-policy** ***sql-statement*** **=> coercion-policy**

Arguments

sql-statement An instance of **<sql-statement>**.

coercion-policy An instance of **<coercion-policy>**.

Description — Returns the coercion policy for *sql-statement*. This method is only relevant to SQL **SELECT** statements.

coercion-policy-setter *Method*

Summary — Sets the coercion policy slot of an SQL statement.

Signature — **coercion-policy-setter** ***new-coercion-policy sql-statement*** **=>** ***new-coercion-policy***

Arguments

new-coercion-policy — An instance of **<coercion-policy>**.

sql-statement — An instance of **<sql-statement>**.

Description — Sets the **coercion-policy** slot of *sql-statement* to *new-coercion-policy*.

datatype-hints *Method*

Summary — Gets an SQL statement's datatype hints.

Signature — **datatype-hints** ***sql-statement*** **=>** ***datatype-hints***

Arguments — *sql-statement* — An instance of **<sql-statement>**.

Values — *datatype-hints* — An instance of **false-or(<sequence>)**.

Description — Gets *sql-statement*'s datatype hints.

datatype-hints-setter *Method*

Summary — Sets an SQL statement's datatype hints.

Signature **datatype-hints-setter** ***new-datatype-hints sql-statement*** **=>** ***new-datatype-hints***

Arguments *new-datatype-hints* An instance of **<object>**.

sql-statement An instance of **<sql-statement>**.

Description Sets the **datatype-hints** slot in *sql-statement* to *new-datatype-hints*.

execute *Generic function*

Summary Prepares an SQL statement for execution on the specified connection and then executes the statement.

Signature **execute** ***sql-statement*** **#key** ***connection parameters result-set-policy liaison*** **=>** ***result-set***

Arguments

sql-statement	An instance of **<sql-statement>**.
connection	An instance of **<connection>**.
parameters	An instance of **false-or(<sequence>)**.
result-set-policy	An instance of **false-or(<result-set-policy>)**.
liaison	An instance of **false-or(<function>)** whose signature is **liaison(<record>)** => **<object>**. Default value: **default-liaison**.

Values *result-set* An instance of **false-or(<result-set>)**.

Description Prepares the SQL statement represented by *sql-statement* for execution on the *connection*, then sends it to the DBMS for execution.

If *connection* is not supplied, **execute** uses the connection returned by **default-connection** instead.

The *liaison* function is invoked on each record as it is retrieved from the database. If a liaison function is not provided, a default value of **`default-liaison`** is used; each result-set class has its own **`default-liaison`**.

In the SQL-ODBC library, the *database-statement* will be an instance of **`<sql-statement>`**. If anonymous host variables—that is, question marks (?)—appear in *database-statement*, pass suitable substitution values in the call to this function.

Example:

This example executes two SQL statements against the database represented by **`the-connection`**. The first SQL statement inserts a new book record into the book table. The second SQL statement queries for the list of titles and their ISBN published by Addison Wesley.

```
with-connection(the-connection)
  let insert-stmt :: <sql-statement> =
  make(<sql-statement>,
    text: "insert into book (title, publisher, isbn)
                values (?, ?, ?)",
    input-indicator: $null-value);
  execute(insert-stmt,
          parameters: #("Large Databases", "Addison-Wesley",
                        $null-value));

let query-stmt :: <sql-statement> =
  make(<sql-statement>,
  text: "select title, isbn from book
              where publisher = ?",
              output-indicator: $null-value);
  execute(query-stmt, parameters: #("Addison-Wesley"));
end with-connection;

=> #(#("An Introduction to Database Systems", "0-201-14201-5"),
     #("Relational Database Writings, 1991-1994", "0-8053-1748-
1), #("Large Databases", $null-value))
```

input-indicator *Method*

Summary Returns the input indicator for an SQL statement.

Signature **`input-indicator`** ***sql-statement*** **`=>`** ***input-indicator***

Arguments *sql-statement* An instance of **`<sql-statement>`**.

Values *input-indicator* An instance of **`<object>`**.

Description Returns the input indicator for *sql-statement*.

input-indicator-setter *Method*

Summary Sets the input indicator of an SQL statement.

Signature **`input-indicator-setter`** ***new-input-indicator sql-statement*** **`=>`** ***new-input-indicator***

Arguments *new-input-indicator* An instance of **`<object>`**.

sql-statement An instance of **`<sql-statement>`**.

Description Sets the **`input-indicator`** slot of *sql-statement* to *new-input-indicator*.

output-indicator *Method*

Summary Returns the output indicator for an SQL statement.

Signature **`output-indicator`** ***sql-statement*** **`=>`** ***output-indicator***

Arguments *sql-statement* An instance of **`<sql-statement>`**.

Values *output-indicator* An instance of **`<object>`**.

Description Returns the output indicator for *sql-statement*.

output-indicator-setter *Method*

Summary Sets the output indicator slot of an SQL statement.

Signature **`output-indicator-setter`** ***new-output-indicator sql-statement***
`=>` ***new-output-indicator***

Arguments *new-output-indicator*
An instance of **`<object>`**.

sql-statement An instance of **`<sql-statement>`**.

Description Sets the **`output-indicator`** slot of *sql-statement* to *new-output-indicator.*

text *Method*

Summary Returns a string containing the text of an SQL statement.

Signature **`text`** ***sql-statement*** **`=>`** ***text***

Arguments *sql-statement* An instance of **`<sql-statement>`**.

Values *text* An instance of **`<string>`**.

Description Returns a string containing the text of *sql-statement*.

text-setter *Method*

Summary Changes the text of an SQL statement.

Signature **`text-setter`** ***new-text sql-statement*** **`=>`** ***new-text***

Arguments *new-text* An instance of **`<string>`**.

sql-statement An instance of **`<sql-statement>`**.

Description Changes the text of the SQL statement in *sql-statement* to *new-text*.

1.4 Data retrieval using result-set collection

Executing an SQL **SELECT** statement by invoking the **execute** function on the instance of **<sql-statement>** that represents the statement yields a result set.

A result set is a Dylan collection which encapsulates the protocol necessary to retrieve data from a database. The SQL-ODBC library defines two subclasses of **<result-set>** that provide different behaviors and performance characteristics. The type of the result set returned by the **execute** function is determined by the result-set *policy* supplied to the function or macro.

There are two subclasses of **<result-set>**: **<forward-only-result-set>** and **<scrollable-result-set>**.

The **<forward-only-result-set>** class provides an efficient means of accessing the elements of a result set. Efficiency is achieved by performing minimal processing on each record retrieved and by maintaining in memory only the current record. Implicit in this behavior is that records you have accessed previously are no longer available to your application; if you maintain references to previous records behavior is unpredictable. The key for each access must always be greater than or equal to the previous access's key; otherwise, a condition is signaled.

The **<scrollable-result-set>** class allows your application to access elements of the result-set collection in any order, meaning that records you have accessed previously can be revisited. Scrollable result sets retrieve records synchronously.

Example:

This example returns a list of authors who have published two or more books.

```
(result-set-policy: make(<scrollable-result-set-policy>))
    select last_name, first_name, count(*)
    from author, book_author
    where book_author.author_id = author.author_id
    group by last_name, first_name
    having count(*) > 2
  end;

=> #(#("Date", "Chris", 2))

let query = make(<sql-statement>,
                text: "select last_name, first_name, count(*)"
                      "from author, book_author"
                      "where book_author.author_id
                         = author.author_id"
                      "group by last_name, first_name having
                         count(*) >= 2");
execute(query, result-set-policy: $scrollable-result-set-policy);
```

1.5 Result-set collections

A result-set collection, in spirit, contains the result of an SQL **SELECT** statement. To provide these results, result-set collections and their methods control the retrieval of elements from the database. Each element of a result set is a record and each element of a record is a value. The SQL-ODBC library does not provide any classes to represent columns; the elements of a record are just Dylan objects.

Result-set classes, in conjunction with the methods defined on them, provide a protocol to retrieve data from a database. Result-sets do not necessarily contain the records from the database. A result set could cache a small subset of the records retrieved for performance reasons. The logic for retrieving a record from a result set (from the database) is as follows:

1. Perform an internal fetch: values are stored into bindings established during SQL statement preparation. A record object is created during the preparation of the SQL statement which represents the values of the record (collection of values).
2. Invoke the liaison method on the record object. The result of the liaison function is the result of the collection access.

The columns of a record are processed when the columns are retrieved from the record object. This includes checking for null values and performing data coercion if a **`coercion-policy`** is supplied.

1.5.1 Record class

An instance of the **`<record>`** class is a placeholder for records retrieved from the database. The record class is a collection whose elements are the columns of the records retrieved from the database. If the record object has a coercion policy (obtained through the **`result-set-policy`**), datatype coercion is performed on the elements of the record object as they are retrieved from the collection.

The elements of a record collection are ephemeral under the result-set retrieval protocol: the values for the elements of the collection can change when the next record of the result set is accessed. A result set may maintain more than one record object to improve performance.

Record collections support the forward- and backward-iteration protocols. The result of calling **`type-for-copy`** on the **`<record>`** class is **`<simple-object-vector>`**.

Applications cannot instantiate the **`<record>`** class. However, the functions returned by the forward- and backward-iteration protocol methods on the result-set classes return instances of this class.

The values in a record object have a short lifespan: they are only valid until the next fetch is performed.

<coercion-policy> *Variable*

Summary Determines what data coercion is to be performed on a result set.

Signature

```
type-union(singleton($default-coercion),
           singleton($no-coercion),
           <sequence>,
           <object>)
```

Description — Determines what data coercion is to be performed on a result set.

<record> *Abstract class*

Summary — The class of records retrieved from a DBMS table as the result of executing an SQL **SELECT** statement.

Superclass — **<sequence>**

Description — The class of records retrieved from a DBMS table as the result of executing an SQL **SELECT** statement.

Instances of this class represent a record that was retrieved from a DBMS table as the result of executing an SQL **SELECT** statement.

If the value passed to **coercion-policy:** is a sequence whose size is less than the degree of the record, the extra columns are converted to their equivalent Dylan type using the default coercion. If the size of the sequence is greater than the degree of the record, the extra elements of the sequence are ignored.

1.5.2 Result-set policy class

Applications use result-set policy classes to specify the behavior and performance characteristics of a result set, as well as its type. The type of the result set is determined by the result-set policy object. The type of the record object is determined by the **coercion-policy** slot of **<sql-statement>**.

If **result-set-policy.scrollable?** is **#t**, the result set will be an instance of **<scrollable-result-set>** otherwise it will be an instance of **<forward-only-result-set>**. If **statement.coercion-policy ~= $no-coercion** then the record will be an instance of **<coercion-record>**; otherwise, it will be an instance of **<record>**.

Scrollable?	Coercion policy	Result set class
#f	#f	<forward-only-result-set>
#t	-	<scrollable-result-set>

Table 1.5 Result set policies and classes.

<result-set-policy> *Concrete class*

Summary: Specifies the behavior and performance characteristics of a result set.

Superclass: **<object>**

Init-keywords:

rowset-size: An instance of **union(<integer>, #"all")**.

scrollable?: An instance of **<boolean>**. Default value: **#f**.

scroll-window: An instance of **<integer>**. A cache size hint.

Description: Specifies the behavior and performance characteristics of a result set.

The **rowset-size** slot is the number of records to retrieve each time an internal fetch is performed. If **rowset-size** is **#"all"**, all records are retrieved the first time a fetch is performed. Currently, rowset-size is ignored.

1.5.3 Result-set classes

Result-sets are the focal point for the encapsulation of the protocol required to retrieve records from a database. The SQL-ODBC library provides three result-set classes with different performance and behavioral characteristics. These classes are **`<result-set>`**, **`<forward-only-result-set>`**, and **`<scrollable-result-set>`**.

<result-set> *Abstract class*

Summary The class for result sets.

Superclasses: **`<sequence>`**

Description Instances of this class represent the results of an SQL **`SELECT`** statement.

This class is the root class for all result-set classes. The **`type-for-copy`** function returns **`<simple-object-vector>`** for objects of this class.

<forward-only-result-set> *Abstract class*

Summary The class for result sets that support a one-shot forward iteration protocol.

Superclasses: **`<result-set>`**

Description Instances of this class represent the results of an SQL **`SELECT`** statement, and support a one-shot forward-iteration-protocol. By one-shot, we mean each element of the collection can be visited only once, and no previously visited element can be revisited. A condition is signaled if the application tries to revisit a record. Thus, backward-iteration-protocol is not supported on this collection.

This collection class is useful when the result of a query is large and each element can be processed individually.

The function **`type-for-copy`** returns **`<simple-object-vector>`** when applied to objects of this class.

<scrollable-result-set> *Abstract class*

Summary
The class for result sets that support both forward and backward iteration.

Superclasses: **`<result-set>`**

Description
Instances of this class support both the forward- and backward-iteration-protocol.

The function **`type-for-copy`** returns **`<simple-object-vector>`** when applied to objects of this class.

1.5.4 Liaison functions

Liaison functions convert records retrieved from a database query to Dylan objects. These functions bridge the conceptual gap between relational databases and Dylan's object-orientation.

To create a Dylan object from a retrieved record, the liaison function must understand the form of the records coming from the database and the mappings of records to Dylan objects. These Dylan objects make up the elements of the result set: the results of the liaison function are added to the result set each time it is called. As your application iterates over a result set, the liaison function provides the objects that the application processes.

If you do not provide a liaison function for a result set, the SQL-ODBC library supplies a **`default-liaison`** function to perform the conversion. If a coercion policy is provided, the **`default-liaison`** function is **`copy-sequence`**. The new sequence is safe in that it is a normal Dylan collection with no relationship to databases, SQL statements, or result sets. If a coercion policy is not provided, the **`default-liaison`** is the identity function.

You can specify the identity function as the liaison function to process the actual record objects. If no type coercion is performed by the functions on the record class, this function will have the lowest overhead, but there are some restrictions: the values retrieved from the record may become invalid when the state of the iteration protocol changes.

The liaison function can, potentially, cause the greatest number of problems for an application using SQL-ODBC since there is no type safety between the liaison function, the record class and the SQL **SELECT** statement. You must ensure that the liaison function is in sync with the SQL **SELECT** statement since there is no support in SQL-ODBC for this.

Example:

```
define class <book> (<object>)
  slot title :: <string>, init-keyword: title:;
  slot publisher :: <string>, init-keyword: publisher:;
  slot isbn :: <string>, init-keyword: isbn:;
  slot author :: <string>, init-keyword: author:;
end class;

begin
  let booker =
    method (record :: <record>) => (book :: <book>)
      let (title, publisher, isbn, last_name, first_name) =
        apply(values, record);

      make(<book>, title: title, publisher: publisher,
           isbn: isbn, author: concatenate(last_name, ", ",
           first_name));
    end method;
let query = make(<sql-statement>,
                 statement: "select title, publisher, isbn,
                               last_name, first_name
                             from book, author, book_author
                             where book.isbn = book_author.isbn
                               and book_author.author_id =
                                     author.author_id
                             order by author.last_name,
                                      author.first_name");
execute(query, liaison: booker
        result-set-policy:
          make(<forward-only-result-set-policy>));
end;
```

1.5.5 Coercion policies

In the SQL-ODBC library, the **`element`** method on the record class encapsulates all coercions of data retrieved from a database. This method can return columns with or without coercion: as low-level SQL data-types (no conversion), as Dylan data-types, or as user-defined types. The **`coercion-policy:`** init-keyword of the **`<sql-statement>`** class determines this behavior.

If the **`coercion-policy:`** init-keyword is **`$no-coercion`**, coercions are not performed. Hence, your application will be processing objects with low-level SQL datatypes. This option has the lowest overhead but the most restrictions: the values returned from the **`element`** method may not be valid (values may change as storage may be reused) after the next call to the **`next-state`** method returned by **`forward-iteration-protocol`**.

The value of **`$default-coercion`** for the **`coercion-policy:`** init-keyword (the default value) indicates that default coercion should be performed: the data retrieved from the database is coerced to the corresponding Dylan objects.

A sequence for the **`coercion-policy:`** init-keyword instructs the SQL library to perform specific data coercion on the data retrieved from the database. Essentially, each element of the limited sequence is a data coercion function which will be invoked using the objects returned from the database as the argument.

When there is a one-to-one correspondence between an SQL datatype and a built-in or user-defined Dylan datatype, use the **`<record>`** class to perform the conversion. When multiple columns define a Dylan object or one column defines multiple Dylan objects, use the liaison function to perform the conversion.

1.6 Data types and conversions

The datatypes that relational DBMSes use are different from those Dylan uses. The SQL-ODBC library provides classes that represent these low-level relational datatypes, along with a table that defines the mapping from these datatypes to Dylan datatypes (Table 1.6, page 36). The methods on the record class consult this mapping when performing data coercion.

The datatypes of host variables are limited to the Dylan datatypes that appear in Table 1.6. Host variables come in two flavors: read and write. Host variables appearing in an into clause of an SQL **SELECT** statement are *write* parameters, and all other host variables are *read* parameters.

DBMS type	SQL type	Dylan type
sql_char	**<sql-char>**	**<character>**
sql_varchar	**<sql-varchar>**	**<string>**
sql_longvarchar	**<sql-longvarchar>**	**<string>**
sql_decimal	**<sql-decimal>**	**<string>**
sql_numeric	**<sql-numeric>**	**<string>**
sql_bit	**<sql-bit>**	**<integer>**
sql_tinyint	**<sql-tinyint>**	**<integer>**
sql_smallint	**<sql-smallint>**	**<integer>**
sql_integer	**<sql-integer>**	**<integer>**
sql_bigint	**<sql-bigint>**	**<integer>**
sql_real	**<sql-real>**	**<single-float>**
sql_float	**<sql-float>**	**<single-float>, <double-float>** or **<extended-float>**
sql_double	**<sql-double>**	**<double-float>**
sql_binary	**<sql-binary>**	**<binary>**
sql_varbinary	**<sql-varbinary>**	**<binary>**
sql_longvarbinary	**<sql-longvarbinary>**	**<binary>**
sql_date	**<sql-date>**	**<date>**
sql_time	**<sql-time>**	**<time>**

Table 1.6 Mapping from DBMS to Dylan datatypes

DBMS type	SQL type	Dylan type
sql_timestamp	**<sql-timestamp>**	**<timestamp>**

Table 1.6 Mapping from DBMS to Dylan datatypes

To retrieve integer elements from databases that may contain more than 30-bit data, you must use the generic-arithmetic library or a run-time error will occur. The Dylan SQL-ODBC library must also be prepared.

Example library and module definition:

```
define library sql-example
  use harlequin-dylan;
  use generic-arithmetic;
  use sql-odbc;

  export sql-example;
end library;

define module sql-example
  use generic-arithmetic-harlequin-dylan;
  use sql-odbc;
end module;
```

1.7 Warning and error conditions

The SQL-ODBC library defines condition classes for each category of error and warning defined in SQL-92. (SQL-92 calls them classes rather than categories.)

When an error or warning occurs, SQL-ODBC detects it, creates a condition object, and signals it. You can then handle the condition using the Dylan condition system.

Some DBMSes can detect and report multiple errors or warnings during the execution of a single SQL statement. The DBMS reports these errors and warnings to the SQL-ODBC library using SQL-92's concept of *diagnostics*; the first error or warning in the diagnostic area is the same error or warning indicated by the **SQLSTATE** status parameter. The SQL-ODBC library signals a condition which corresponds to the error or warning indicated by **SQLSTATE**.

While handling the first condition, your application can process any additional errors or warnings that may have occurred by signaling the next DBMS condition; to obtain the next DBMS condition, call **`next-dbms-condition`** on the condition being handled.

1.7.1 Diagnostics

SQL-92 defines a *diagnostics area* as a DBMS-managed data structure that captures specific information about the execution of a SQL statement, with the exception of the **`GET DIAGNOSTICS`** statement. A diagnostics area consists of two sections, a header and a collection of diagnostic details.

The header contains information about the last SQL statement executed, while the diagnostic details contain information about each error or warning that resulted from the execution of the SQL statement.

The size of the diagnostic details section is the default value for the DBMS implementation. This size is always greater than one, since the first diagnostic detail corresponds to **`sqlstate`**. A DBMS may only fill in one diagnostic detail regardless of the number of errors or warnings that occur. If multiple diagnostic details are filled in, there is no presumption of precedence or importance.

The SQL-ODBC library provides wrapper classes for these constructs and accessors for the information they represent.

row-count *Generic function*

Summary Returns the number of rows that were affected by the last SQL statement to be executed.

Signature **`row-count`** ***diagnostic*** **`=>`** ***count***

Arguments *diagnostic* An instance of **`<diagnostic>`**.

Values *count* An instance of **`<integer>`**.

Description Returns the number of rows that were affected by the last SQL statement to be executed.

<diagnostic> *Abstract class*

Summary The class that encapsulates diagnostic information returned by the DBMS.

Superclasses: **`<condition>`**

Description Encapsulates all diagnostic information returned by a DBMS, including diagnostic header and diagnostic records, as defined by SQL-92.

condition-number *Generic function*

Summary Returns the element key.

Signature **`condition-number`** ***diagnostic*** **`=>`** ***condition-number***

Arguments *diagnostic* An instance of **`<diagnostic>`**.

Values *condition-number* An instance of **`<integer>`**.

Description Returns the element key.

returned-sqlstate *Generic function*

Summary Returns the SQL state that corresponds to the reported error or warning.

Signature **`returned-sqlstate`** ***diagnostic*** **`=>`** ***sqlstate***

Arguments *diagnostic* An instance of **`<diagnostic>`**.

Values *sqlstate* An instance of **`limited(<string>, size: 5)`**.

Description
: Returns the **`sqlstate`** that corresponds to the error or warning reported in the detail area.

class-origin *Generic function*

Summary
: Returns ISO 9075 if the class code value is defined in the SQL-92 standard.

Signature
: **`class-origin`** ***diagnostic*** **`=>`** ***class-origin***

Arguments
: *diagnostic* An instance of **`<diagnostic>`**.

Values
: *class-origin* An instance of **`<string>`**.

Description
: Returns **`"ISO 9075"`** if the class code value is defined in the SQL-92 standard. Otherwise, the value of *class-origin* will depend on the DBMS.

subclass-origin *Generic function*

Summary
: Returns ISO 9075 if the subclass code value is defined in the SQL-92 standard.

Signature
: **`subclass-origin`** ***diagnostic*** **`=>`** ***subclass-origin***

Arguments
: *diagnostic* An instance of **`<diagnostic>`**.

Values
: *subclass-origin* An instance of **`<string>`**.

Description
: Returns **`"ISO 9075"`** if the subclass code value is defined in the SQL-92 standard. Otherwise, the value of *subclass-origin* will depend on the DBMS.

connection-name

Generic function

Summary
: Returns the name of the connection that was used to execute the SQL statement.

Signature
: **`connection-name`** ***diagnostic*** **`=>`** ***connection-name***

Arguments
: *diagnostic* An instance of **`<diagnostic>`**.

Values
: *connection-name* An instance of **`<string>`**.

Description
: Returns the name of the connection that was used to execute the SQL statement.

message-text

Generic function

Summary
: Returns a text string containing a natural-language error text if a DBMS supplies one.

Signature
: **`message-text`** ***diagnostic*** **`=>`** ***message-text***

Arguments
: *diagnostic* An instance of **`<diagnostic>`**.

Values
: *message-text* An instance of **`<string>`**.

Description
: Returns a text string containing a natural-language error text if a DBMS supplies one. Otherwise it returns the empty string. A DBMS is not required to supply this information.

1.7.2 SQL condition classes

Below, the exact class code or subclass code, as defined by SQL-92, is listed with its Dylan implementation class.

<ambiguous-cursor-name> *Open abstract class*

Superclasses `<diagnostic>`

Class-code “3C”

<cardinality-violation> *Open abstract class*

Superclasses `<diagnostic>`

Class-code “21”

<connection-exception> *Open abstract class*

Superclasses `<diagnostic>`

Class-code “08”

<connection-does-not-exist> *Open abstract class*

Superclasses `<connection-exception>`

Class-code “003”

<connection-failure> *Open abstract class*

Superclasses `<connection-exception>`

Class-code “006”

<connection-name-in-use> *Open abstract class*

Superclasses **`<connection-exception>`**

Class-code “002”

<sql-client-unable-to-establish-connection> *Open abstract class*

Superclasses **`<connection-exception>`**

Class-code “001”

<sql-server-rejected-establishment-of-connection>

Open abstract class

Superclasses **`<connection-exception>`**

Class-code “004”

<transaction-resolution-unknown> *Open abstract class*

Superclasses **`<connection-exception>`**

Class-code “007”

<cursor-operation-conflict> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “09”

<data-exception> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "22"

<character-not-in-repertoire> *Open abstract class*

Superclasses `<data-exception>`

Subclass-code "021"

<datetime-field-overflow> *Open abstract class*

Superclasses `<data-exception>`

Subclass-code "008"

<division-by-zero> *Open abstract class*

Superclasses `<data-exception>`

Subclass-code "012"

<error-in-assignment> *Open abstract class*

Superclasses `<data-exception>`

Subclass-code “005”

<indicator-overflow> *Open abstract class*

Superclasses **`<data-exception>`**

Subclass-code “022”

<interval-field-overflow> *Open abstract class*

Superclasses **`<data-exception>`**

Subclass-code “015”

<invalid-character-value-for-cast> *Open abstract class*

Superclasses **`<data-exception>`**

Subclass-code “018”

<invalid-datetime-format> *Open abstract class*

Superclasses **`<data-exception>`**

Subclass-code “007”

<invalid-escape-character> *Open abstract class*

Superclasses **`<data-exception>`**

Subclass-code “019”

<invalid-escape-sequence> *Open abstract class*

Superclasses **`<data-exception>`**

Subclass-code “025”

<invalid-fetch-sequence> *Open abstract class*

Superclasses **`<data-exception>`**

Subclass-code “006”

<invalid-parameter-value> *Open abstract class*

Superclasse **`<data-exception>`**

Subclass-code “023”

<invalid-time-zone-displacement-value> *Open abstract class*

Superclasses **`<data-exception>`**

Subclass-code	“009”

<null-value-no-indicator-parameter> *Open abstract class*

Superclasses	**`<data-exception>`**
Subclass-code	“002”

<Numeric-value-out-of-range> *Open abstract class*

Superclasses	**`<data-exception>`**
Subclass-code	“003”

<string-data-length-mismatch> *Open abstract class*

Superclasses	**`<data-exception>`**
Subclass-code	“026”

<string-data-right-truncation> *Open abstract class*

Superclasses	**`<data-exception>`**
Subclass-code	“001”

<substring-error> *Open abstract class*

Superclasses `<data-exception>`

Subclass-code “011”

<trim-error> *Open abstract class*

Superclasses `<data-exception>`

Subclass-code “027”

<unterminated-C-string> *Open abstract class*

Superclasses `<data-exception>`

Subclass-code “024”

<dependent-privilege-descriptors-still-exist> *Open abstract class*

Superclasses `<diagnostic>`

Class-code “2B”

<dynamic-sql-error> *Open abstract class*

Superclasses `<diagnostic>`

Class-code “07”

<cursor-specification-cannot-be-executed> *Open abstract class*

Superclasses **`<dynamic-sql-error>`**

Subclass-code “003”

<invalid-descriptor-count> *Open abstract class*

Superclasses **`<dynamic-sql-error>`**

Subclass-code “008”

<invalid-descriptor-index> *Open abstract class*

Superclasses **`<dynamic-sql-error>`**

Subclass-code “009”

<prepared-statement-not-a-cursor-specification> *Open abstract class*

Superclasses **`<dynamic-sql-error>`**

Subclass-code “005”

<restricted-data-type-attribute-violation> *Open abstract class*

Superclasses **`<dynamic-sql-error>`**

Subclass-code “006”

<using-clause-does-not-match-dynamic-parameter-specification>

Open abstract class

Superclasses **`<dynamic-sql-error>`**

Subclass-code “001”

<using-clause-does-not-match-target-specification>

Open abstract class

Superclasses **`<dynamic-sql-error>`**

Subclass-code “002”

<using-clause-required-for-dynamic-parameters> *Open abstract class*

Superclasses **`<dynamic-sql-error>`**

Subclass-code “004”

<using-clause-required-for-result-fields> *Open abstract class*

Superclasses **`<dynamic-sql-error>`**

Subclass-code “007”

<feature-not-supported> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “0A”

<multiple-server-transaction> *Open abstract class*

Superclasses **`<feature-not-supported>`**

Subclass-code “001”

<integrity-constraint-violation> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “23”

<invalid-authorization-specification> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “28”

<invalid-catalog-name> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “3D”

<invalid-character-set-name> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “2C”

<invalid-condition-number> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “35”

<invalid-cursor-name> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “34”

<invalid-schema-name> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “3F”

<invalid-sql-descriptor-name> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “33”

<invalid-sql-statement-name> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "26"

<invalid-transaction-state> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "25"

<invalid-transaction-termination> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "2D"

<no-data> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "02"

<remote-database-access> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "HZ"

<successful-completion> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "00"

<syntax-error-or-access-rule-violation> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "42"

<syntax-error-or-access-rule-violation-in-direct-sql-statement> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "2A"

<syntax-error-or-access-rule-violation-in-dynamic-sql-statement> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "37"

<transaction-rollback> *Open abstract class*

Superclasses `<diagnostic>`

Class-code "40"

<transaction-rollback-due-to-integrity-constraint-violation> *Open abstract class*

Superclasses **`<transaction-rollback>`**

Subclass-code "002"

<transaction-rollback-due-to-serialization-failure> *Open abstract class*

Superclasses **`<transaction-rollback>`**

Subclass-code "001"

<statement-completion-unknown> *Open abstract class*

Superclasses **`<transaction-rollback>`**

Subclass-code "003"

<triggered-data-change-violation> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code "27"

`SQL`

Module

<sql-warning> *Open abstract class*

Superclasses **`<diagnostic>`**

Class-code “01”

<warning-cursor-operation-conflict> *Open abstract class*

Superclasses **`<sql-warning>`**

Subclass-code “001”

<disconnect-error> *Open abstract class*

Superclasses **`<sql-warning>`**

Subclass-code “002”

<implicit-zero-bit-padding> *Open abstract class*

Superclasses **`<sql-warning>`**

Subclass-code “008”

<insufficient-item-descriptor-areas> *Open abstract class*

Superclasses	**`<sql-warning>`**
Subclass-code	"005"

<null-value-eliminated-in-set-function> *Open abstract class*

Superclasses	**`<sql-warning>`**
Subclass-code	"003"

<privilege-not-granted> *Open abstract class*

Superclasses	**`<sql-warning>`**
Subclass-code	"007"

<privilege-not-revoked> *Open abstract class*

Superclasses	**`<sql-warning>`**
Subclass-code	"006"
Library	**`SQL`**
Module	**`SQL`**

<query-expression-too-long-for-information-schema>

Open abstract class

Superclasses `<sql-warning>`

Subclass-code "00A"

<search-condition-too-long-for-information-schema>

Open abstract class

Superclasses `<sql-warning>`

Subclass-code "009"

<warning-string-data-right-truncation>

Open abstract class

Superclasses `<sql-warning>`

Subclass-code "004"

<with-check-option-violation>

Open abstract class

Superclasses `<diagnostic>`

Class-code "44"

1.7.3 ODBC-specific extensions to the diagnostic protocol

native-error-code *Function*

Summary Returns a driver / data source-specific native error code.

Signature **`native-error-code`** ***diagnostic*** **`=>`** ***error-code***

Arguments *diagnostic* An instance of **`<odbc-diagnostic>`**.

Values *error-code* An instance of **`<integer>`**.

Description Returns a driver / data source-specific native error code. If there is no native error code, it returns 0.

column-number *Function*

Summary Returns a value that represents the column number in the result set.

Signature **`column-number`** ***diagnostic*** **`=>`** ***column-number***

Arguments *diagnostic* An instance of **`<odbc-diagnostic>`**.

Values *column-number* An instance of **`<integer>`**.

Description If the result of **`row-number`** is a valid row number in a result set, this function returns a value that represents the column number in the result set. Column numbers begin at 1 rather than 0.

row-number *Function*

Summary Returns the row number at which the diagnostic occurred.

Signature **`row-number`** ***diagnostic*** **`=>`** ***row-number***

Arguments *diagnostic* An instance of **`<odbc-diagnostic>`**.

row-number An instance of **`<integer>`**.

Description Returns the row number at which *diagnostic* occurred.

1.7.4 ODBC-specific diagnostic classes

The following diagnostic classes are unique to ODBC.

<odbc-invalid-connection-string-attribute> *Sealed concrete class*

Superclasses **`<odbc-sql-warning>`**

Class-code "01"

Subclass-code "S00"

<odbc-error-in-row> *Sealed concrete class*

Superclasses **`<odbc-sql-warning>`**

Class-code "01"

Subclass-code "S01"

<odbc-option-value-changed> *Sealed concrete class*

Superclasses **`<odbc-sql-warning>`**

Class-code	“01”
Subclass-code	“S02”

<odbc-attempt-to-fetch-before-result-set-returned-the-first-rowset>

Sealed concrete class

Superclasses	**`<odbc-sql-warning>`**
Class-code	“01”
Subclass-code	“S06”

<odbc-fractional-truncation>

Sealed concrete class

Superclasses	**`<odbc-sql-warning>`**
Class-code	“01”
Subclass-code	“S07”

<odbc-error-saving-file-dsn>

Sealed concrete class

Superclasses	**`<odbc-sql-warning>`**
Class-code	“01”
Subclass-code	“S08”

<odbc-invalid-keyword> *Sealed concrete class*

Superclasses **`<odbc-sql-warning>`**

Class-code “01”

Subclass-code “S09”

<odbc-invalid-use-of-default-parameter> *Sealed concrete class*

Superclasses **`<odbc-dynamic-sql-error>`**

Class-code “07”

Subclass-code “S01”

<odbc-communication-link-failure> *Sealed concrete class*

Superclasses **`<odbc-connection-exception>`**

Class-code “08”

Subclass-code “S01”

<odbc-insert-value-list-does-not-match-column-list>

Sealed concrete class

Superclasses **`<odbc-cardinality-violation>`**

Class-code “21”

Subclass-code "S01"

<odbc-invalid-cursor-state> *Sealed concrete class*

Superclasses **`<odbc-diagnostic>`**

Class-code "24"

Subclass-code "000"

<odbc-transaction-state>

Sealed concrete class

Superclasses **`<odbc-invalid-transaction-state>`**

Class-code "25"

Subclass-code "S01"

<odbc-transaction-still-active> *Sealed concrete class*

Superclasses **`<odbc-invalid-transaction-state>`**

Class-code "25"

Subclass-code "S02"

<odbc-transaction-is-rolledback> *Sealed concrete class*

Superclasses **`<odbc-invalid-transaction-state>`**

Class-code "25"

Subclass-code "S03"

<odbc-base-table-or-view-already-exists> *Sealed concrete class*

Superclasses **`<odbc-syntax-error-or-access-rule-violation>`**

Class-code "42"

Subclass-code "S01"

<odbc-base-table-or-view-not-found> *Sealed concrete class*

Superclasses **`<odbc-syntax-error-or-access-rule-violation>`**

Class-code "42"

Subclass-code "S02"

<odbc-index-already-exists> *Sealed concrete class*

Superclasses **`<odbc-syntax-error-or-access-rule-violation>`**

Class-code "42"

Subclass-code "S11"

<odbc-index-not-found> *Sealed concrete class*

Superclasses **`<odbc-syntax-error-or-access-rule-violation>`**

Class-code "42"

Subclass-code "S12"

<odbc-column-already-exists> *Sealed concrete class*

Superclasses **`<odbc-syntax-error-or-access-rule-violation>`**

Class-code "42"

Subclass-code "S21"

<odbc-column-not-found> *Sealed concrete class*

Superclasses **`<odbc-syntax-error-or-access-rule-violation>`**

Class-code "42"

Subclass-code "S22"

<odbc-general-error> *Sealed concrete class*

Superclasses `<odbc-diagnostic>`

Class-code "HY"

Subclass-code "000"

<odbc-memory-allocation-error> *Sealed concrete class*

Superclasses `<odbc-diagnostic>`

Class-code "HY"

Subclass-code "001"

<odbc-invalid-application-buffer-type> *Sealed concrete class*

Superclasses `<odbc-diagnostic>`

Class-code "HY"

Subclass-code "003"

<odbc-invalid-sql-data-type> *Sealed concrete class*

Superclasses `<odbc-diagnostic>`

Class-code "HY"

Subclass-code	"004"

<odbc-associated-statement-is-not-prepared> *Sealed concrete class*

Superclasses	`<odbc-diagnostic>`
Class-code	"HY"
Subclass-code	"007"

<odbc-operation-canceled> *Sealed concrete class*

Superclasses	`<odbc-diagnostic>`
Class-code	"HY"
Subclass-code	"008"

<odbc-invalid-use-of-null-pointer> *Sealed concrete class*

Superclasses	`<odbc-diagnostic>`
Class-code	"HY"
Subclass-code	"009"

<odbc-function-sequence-error> *Sealed concrete class*

Superclasses **`<odbc-diagnostic>`**

Class-code "HY"

Subclass-code "010"

<odbc-attribute-cannot-be-set-now> *Sealed concrete class*

Superclasses **`<odbc-diagnostic>`**

Class-code "HY"

Subclass-code "011"

<odbc-invalid-transaction-operation-code> *Sealed concrete class*

Superclasses **`<odbc-diagnostic>`**

Class-code "HY"

Subclass-code "012"

<odbc-memory-management-error> *Sealed concrete class*

Superclasses **`<odbc-diagnostic>`**

Class-code "HY"

Subclass-code	“013”

<odbc-limit-on-the-number-of-handles-exceeded>

Sealed concrete class

Superclasses	`<odbc-diagnostic>`
Class-code	“HY”
Subclass-code	“014”

<odbc-no-cursor-name-available>

Sealed concrete class

Superclasses	`<odbc-diagnostic>`
Class-code	“HY”
Subclass-code	“015”

<odbc-cannot-modify-an-implementation-row-descriptor>

Sealed concrete class

Superclasses:	`<odbc-diagnostic>`
Class-code:	“HY”
Subclass-code:	“016”

<odbc-invalid-use-of-an-automatically-allocated-descriptor-handle>

Sealed concrete class

Superclasses	**`<odbc-diagnostic>`**
Class-code:	HY”
Subclass-code	“017”

<odbc-server-declined-cancel-request>

Sealed concrete class

Superclasses	**`<odbc-diagnostic>`**
Class-code	“HY”
Subclass-code	“018”

<odbc-non-character-and-non-binary-data-sent-in-pieces>

Sealed concrete class

Superclasses	**`<odbc-diagnostic>`**
Class-code	“HY”
Subclass-code	“019”

<odbc-attempt-to-concatenate-a-null-value>

Sealed concrete class

Superclasses	**`<odbc-diagnostic>`**

Class-code "HY"

Subclass-code "020"

<odbc-inconsistent-descriptor-information> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: "HY"

Subclass-code: "021"

<odbc-invalid-attribute-value> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: "HY"

Subclass-code: "024"

<odbc-invalid-string-or-buffer-length> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: "HY"

Subclass-code: "090"

<odbc-invalid-descriptor-field-identifier> *Sealed concrete class*

Superclasses: `<odbc-diagnostic>`

Class-code: "HY"

Subclass-code: "091"

<odbc-invalid-attribute-option-identifier> *Sealed concrete class*

Superclasses: `<odbc-diagnostic>`

Class-code: "HY"

Subclass-code: "092"

<odbc-invalid-parameter-number> *Sealed concrete class*

Superclasses: `<odbc-diagnostic>`

Class-code: "HY"

Subclass-code: "093"

<odbc-function-type-out-of-range> *Sealed concrete class*

Superclasses: `<odbc-diagnostic>`

Class-code: "HY"

Subclass-code: "095"

<odbc-invalid-information-type> *Sealed concrete class*

Superclasses: `<odbc-diagnostic>`

Class-code: "HY"

Subclass-code: "096"

<odbc-column-type-out-of-range> *Sealed concrete class*

Superclasses: `<odbc-diagnostic>`

Class-code: "HY"

Subclass-code: "097"

<odbc-scope-type-out-of-range> *Sealed concrete class*

Superclasses: `<odbc-diagnostic>`

Class-code: "HY"

Subclass-code: "098"

<odbc-nullable-type-out-of-range> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: “HY”

Subclass-code: “099”

<odbc-uniqueness-option-type-out-of-range> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: “HY”

Subclass-code: “100”

<odbc-accuracy-option-type-out-of-range> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: “HY”

Subclass-code: “101”

<odbc-invalid-retrieval-code> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: “HY”

Subclass-code: “103”

<odbc-invalid-precision-or-scale-value> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: “HY”

Subclass-code: “104”

<odbc-invalid-parameter-type> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: “HY”

Subclass-code: “105”

<odbc-fetch-type-out-of-range> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: “HY”

Subclass-code: “106”

<odbc-row-value-out-of-range> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: "HY"

Subclass-code: "107"

<odbc-invalid-cursor-position> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: "HY"

Subclass-code: "109"

<odbc-invalid-driver-completion> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: "HY"

Subclass-code: "110"

<odbc-invalid-bookmark-value> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: "HY"

Subclass-code: “111”

<odbc-optional-feature-not-implemented> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: “HY”

Subclass-code: “C00”

<odbc-timeout-expired> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: “HY”

Subclass-code: “T00”

<odbc-connection-timeout-expired> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: “HY”

Subclass-code: “T01”

<odbc-driver-does-not-support-this-function> *Sealed concrete class*

Superclasses: **`<odbc-diagnostic>`**

Class-code: "IM"

Subclass-code: "001"

<odbc-data-source-name-not-found> *Sealed concrete class*

Superclasses: **`<odbc-connection-exception>`**

Class-code: "IM"

Subclass-code: "002"

<odbc-specified-driver-could-not-be-loaded> *Sealed concrete class*

Superclasses: **`<odbc-connection-exception>`**

Class-code: "IM"

Subclass-code: "003"

<odbc-driver-SQLAllocHandle-on-SQL-HANDLE-ENV-failed> *Sealed concrete class*

Superclasses: **`<odbc-connection-exception>`**

Class-code: "IM"

Subclass-code: "004"

<odbc-driver-SQLAllocHandle-on-SQL-HANDLE-DBC-failed>

Sealed concrete class

Superclasses: **`<odbc-connection-exception>`**

Class-code: “IM”

Subclass-code: “005”

<odbc-driver-SQLSetConnectAttr-failed>

Sealed concrete class

Superclasses: **`<odbc-connection-exception>`**

Class-code: “IM”

Subclass-code: “006”

<odbc-no-data-source-or-driver-specified>

Sealed concrete class

Superclasses: **`<odbc-connection-exception>`**

Class-code: “IM”

Subclass-code: “007”

<odbc-dialog-failed>

Sealed concrete class

Superclasses **`<odbc-connection-exception>`**

Class-code “IM”

Subclass-code:	“008”

<odbc-unable-to-load-translation-dll> *Sealed concrete class*

Superclasses	**`<odbc-connection-exception>`**
Class-code	“IM”
Subclass-code	“009”

<odbc-data-source-name-too-long> *Sealed concrete class*

Superclasses	**`<odbc-connection-exception>`**
Class-code	“IM”
Subclass-code	“010”

<odbc-driver-name-too-long> *Sealed concrete class*

Superclasses	**`<odbc-connection-exception>`**
Class-code	“IM”
Subclass-code	“011”

<odbc-DRIVER-keyword-syntax-error> *Sealed concrete class*

Superclasses **`<odbc-connection-exception>`**

Class-code "IM"

Subclass-code "012"

<odbc-invalid-name-of-file-DSN> *Sealed concrete class*

Superclasses **`<odbc-connection-exception>`**

Class-code "IM"

Subclass-code "014"

<odbc-corrupt-file-data-source> *Sealed concrete class*

Superclasses **`<odbc-connection-exception>`**

Class-code "IM"

Subclass-code "015"

<odbc-trace-file-error> *Sealed concrete class*

Superclasses **`<odbc-diagnostic>`**

Class-code "IM"

Subclass-code	"013"

1.8 Database introspection

The SQL-ODBC library offers introspection features to allow you to determine the structure of a database at run time. A database structure is a hierarchy comprising catalogs, schemas, tables and columns. A catalog is a named collection of schemas, a schema is a named collection of tables, and a table is a named collection of columns. For security reasons, the SQL-ODBC library does not provide any means of obtaining a list of databases available from a particular DBMS; your application must provide access to a particular database via a connection object.

For DBMSes which do not support catalogs or schemas, the SQL-ODBC library uses a default value that your application can use to perform introspection.

1.8.1 Database objects and integrity constraints

You can interrogate schema and table database objects for a collection of constraints defined against them. A *constraint* is a data integrity rule which the DBMS enforces at all times. These constraints are **`unique`**, **`primary key`**, **`referential`** and **`check`**.

The **`unique`** constraint specifies that one or more columns within a table must have a unique value or set of values for each record in the table (however, the set of columns are not necessarily a key). The **`primary`** key constraint is similar to the **`unique`** constraint, except the set of columns must uniquely identify records within the table.

The **`referential`** constraint specifies the relationship between a column or a group of columns in one table to another table; this constraint also specifies the action to take when records within the table are updated or deleted.

Finally, the **`check`** constraint is a general constraint on a table which must never be false and, due to three-valued logic, an unknown or null value will satisfy the constraint.

An **`assertion`** is a constraint on a schema. It is similar to the **`check`** constraint but it normally involves more than one table. The significant difference between an **`assertion`** and a **`check`** is that an **`assertion`** must always be true, whereas a **`check`** must never be false.

The nullability of a column is a column constraint which can be determined by introspection on the desired column.

Syntactically, SQL-92 supports table and column constraints; semantically, however, all constraints are enforced at the table level.

<database-object> *Abstract class*

Superclasses **`<object>`**

Description A common ancestor for all introspection classes. (Not to be used directly by users.)

database-object-name *Generic function*

Summary Returns the name of a database object.

Signature **`database-object-name`** ***db-object*** **`=>`** ***name-string***

Arguments
db-object An instance of **`<database-object>`**.
name-string An instance of **`<string>`**.

Description Returns the name of the database object. It is inherited by the subclasses of **`<database-object>`**.

1.8.2 Catalogs

<catalog> *Abstract class*

Summary: The class of collection objects that contain **`<schema>`** objects.

Superclasses: **`<database-object>`**, **`<result-set>`**

Description: Instances of the **`<catalog>`** class are collection objects each of whose elements is an instance of **`<schema>`**. This result set can be considered the top-level of the hierarchy of introspection objects.

catalogs *Generic function*

Summary: Returns a collection of catalogs associated with a specified connection.

Signature: **`catalogs`** ***connection*** **`=>`** ***collection-of-catalogs***

Arguments:

connection An instance of **`<connection>`**.

collection-of-catalogs An instance of **`<result-set>`**.

Description: Returns a collection of catalogs for the database associated with *connection*. For DBMSes which do not yet support catalogs, the SQL-ODBC library will return a default catalog.

1.8.3 Schema

<schema> *Abstract class*

Summary — The class of collection objects that contain **`<sql-table>`** objects.

Superclasses — **`<database-object>`**, **`<result-set>`**

Description — Instances of this class are collection objects each of whose elements is an instance of **`<sql-table>`**.

1.8.4 Tables

<sql-table> *Abstract class*

Summary — The class of SQL tables.

Superclasses — **`<database-object>`**, **`<result-set>`**

Description — Instances of the **`<sql-table>`** class are collection objects that support the **`forward-iteration-protocol`**. Each element of a table collection is an instance of **`<column>`**.

indexes *Generic function*

Summary — Return a collection of indexes defined on a table.

Signature — **`indexes`** ***table*** **`=>`** ***index-collection***

Arguments

table	An instance of **`<sql-table>`**.
index-collection	An instance of **`<result-set>`**.

Description — Returns a collection of indexes defined on *table*.

1.8.5 Columns

<column> — *Abstract class*

Summary — The class of columns.

Superclasses — **`<database-object>`**

Description — Instances of the **`<column>`** class are objects that represent the columns in an RDBMS table.

domain — *Generic function*

Summary — Returns the domain type of a column.

Signature — **`domain`** ***column*** **`=>`** ***type***

Arguments

column	An instance of **`<column>`**.
type	An instance of **`<sql-type>`**.

Description — Returns the domain type of the column.

nullable? — *Generic function*

Summary — Returns the nullability of a column.

Signature — **`nullable? column => nullable`**

Arguments

column	An instance of **`<column>`**.
nullable	An instance of **`<boolean>`**.

Description — Returns the nullability of a column.

default-value *Generic function*

Summary — Returns the default value of a column.

Signature — **`default-value`** ***column* => *sql-value***

Arguments

column	An instance of **`<column>`**.
sql-value	An instance of **`<object>`**.

Description — Returns the default value of a column, assigned when the column was created. If the DBMS does not support default values, the SQL-ODBC library signals a warning and returns **`$null-value`**.

1.8.6 Indexes

An index is a concept implemented by most DBMS vendors, but is not defined in the SQL-92 standard. The **`CREATE INDEX`** and **`DROP INDEX`** statements are DBMS-specific extensions to the SQL-92 standard.

<index> *Abstract class*

Superclasses — **`<database-object>`**

indexed-table *Generic function*

Summary — Returns the *table* associated with this *index*.

Signature — **`indexed-table`** ***index* => *table***

Arguments

index	An instance of **`<index>`**.

Values	*table*	An instance of **`<sql-table>`**.

Description Returns the indexed table *table*.

fields *Generic function*

Summary Returns a field sequence.

Signature **`fields`** ***index*** **`=>`** ***field-sequence***

Arguments	*index*	An instance of **`<index>`**.
Values	*field-sequence*	An instance of **`<sequence>`**.

Description Returns a field sequence.

unique-index? *Generic function*

Summary Returns **`#t`** if the given index is unique.

Signature **`unique-index?`** ***index*** **`=>`** ***unique***

Arguments	*index*	An instance of **`<index>`**.
Values	*unique*	An instance of **`<boolean>`**.

Description Returns **`#t`** if the given index is unique.

2

Harlequin Dylan's COM, OLE, and ActiveX Libraries

2.1 Introduction

This chapter introduces Harlequin Dylan's support for Microsoft's Component Object Model, COM, and the various technologies based on it—Object Linking and Embedding (OLE) for compound documents, OLE Automation, OLE / ActiveX controls, and so on.

2.1.1 About COM

Microsoft's Component Object Model, COM, is an important component software technology. It is the foundation upon which technologies such as Object Linking and Embedding (OLE), OLE Automation, and OLE / ActiveX controls are built.

COM is a Microsoft standard for specifying and deploying software components so that components can specify their public interfaces independently of the language they are written in. COM is basically a set of rules to which component software should conform.

COM permits this language independence by specifying its own binary protocol for communication between software components. It does not matter what language you implement a component in, as long as it can communicate with other COM components according to COM's binary protocol.

COM technologies generally work on a client-server model, with client and server applications communicating using COM protocols.

The Windows operating systems include COM support libraries that support client-server communication for COM-based software. The Windows Registry also plays a part in the way COM technologies work.

2.1.2 About OLE and compound documents

OLE stands for Object Linking and Embedding. It is a technology built on top of COM that allows the creation of *compound documents*.

Compound documents are basically documents that contain data from more than one application. When viewing the document, you can see and edit the data from both applications. For example, a compound document could be a text document from a word processor that contains a picture from a painting application. In OLE/COM terminology, the word processor is the *container application* and the painting application is the *server application*.

The "Linking and Embedding" part of the OLE name relates to how the object data from the server application is stored in the container document, so that it can be reinstated when the container document is next opened.

In our example, this means how the picture object is stored in the word processor document. If the picture object is *linked*, it is stored in its own file, and the text document contains a link to that file in the appropriate place. If the picture object is *embedded*, it is stored in the same file as the text document.

See Section 2.1.6 for an example of using compound documents.

2.1.3 About OLE Automation

OLE Automation, or Automation, is another technology built on top of COM. Automation is a general mechanism for dynamically communicating data and commands between applications.

Like the OLE compound documents technology, Automation works through a client-server mechanism. But unlike compound documents, the server and client need not offer document-style functionality, or have any on-screen representation at all. Automation simply allows a client application, or *controller*, to access functionality in a server that the server's developer chose to expose.

Automation is commonly used to provide a programmable interface to an application that is chiefly operated by a human user. For example, a word processing application might offer services via Automation that a script could call on to produce formatted reference documentation from raw source code input.

Microsoft Visual Basic is a popular language for writing Automation controllers, and indeed many of the details of OLE Automation were designed to accommodate Visual Basic's needs.

2.1.4 About OLE (ActiveX) controls

OLE controls, also known as ActiveX controls, combine features of OLE Automation and the OLE compound documents technology. A control is a component with a GUI that can be included in a client application or *container*. A control not only has a visual representation in the container, but is also capable of sending data and commands to the container and receiving them from the container.

2.1.5 Terminology

Harlequin Dylan supports OLE 2, and follows Microsoft in referring to it as simply "OLE".

We use "OLE / COM" as an umbrella term for all COM-based technologies.

Some of the technologies built on top of COM are now officially labeled "ActiveX", but we generally use the longer-standing term "OLE" to describe them. Thus we say "OLE control" rather than "ActiveX control".

2.1.6 OLE/COM example: using compound documents

To give an example of OLE / COM technologies in action, we will now create a compound document and give a very high-level overview of what is going on as we do so. Compound documents are the simplest OLE / COM technology to demonstrate, since all basic Windows 95 and NT 4 setups include some applications that can be used as compound document containers and servers. If you have created a compound document before, you may wish to skip this section.

To begin, simply start the WordPad program. It should be available from the **Start** menu as **Programs > Accessories > WordPad**. Type some text into the empty document window. We can embed a picture into this document, thereby creating a compound document.

Choose **Insert > Object** in WordPad. This command brings up the Insert Object dialog, which shows a list of COM objects you can embed in a WordPad document.

This is actually a list of new objects; by changing the option button selection from **Create New** to **Create from File** we can also embed an existing file into the document and activate it using the application that created it.

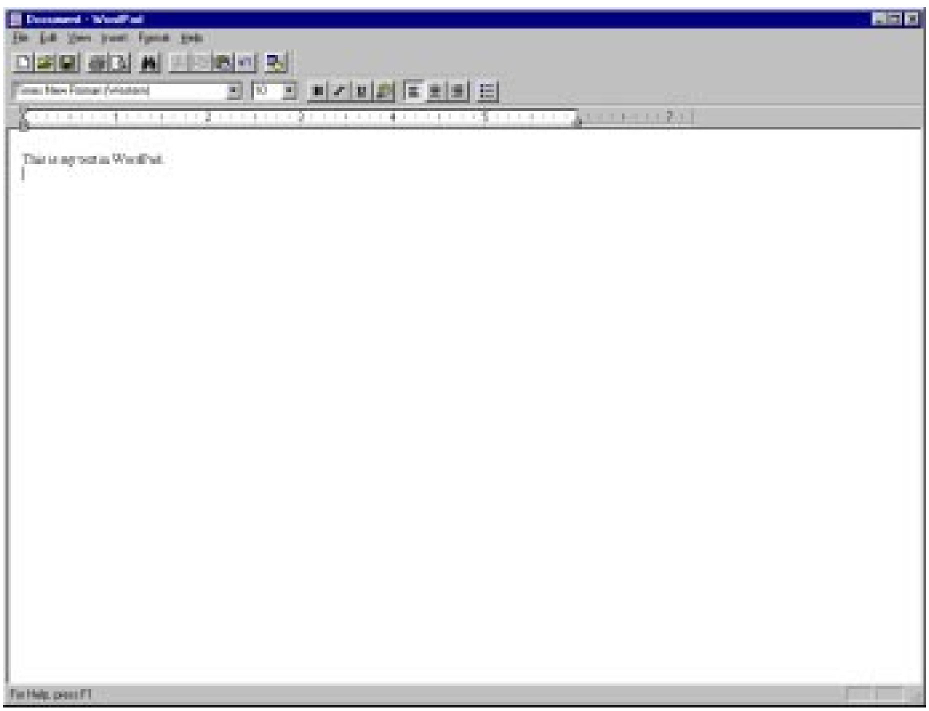

Figure 2.1 A new WordPad document.

For this example, we want to create a new Paintbrush picture. Select **Paintbrush Picture** in the **Object Type** list and click **OK**.

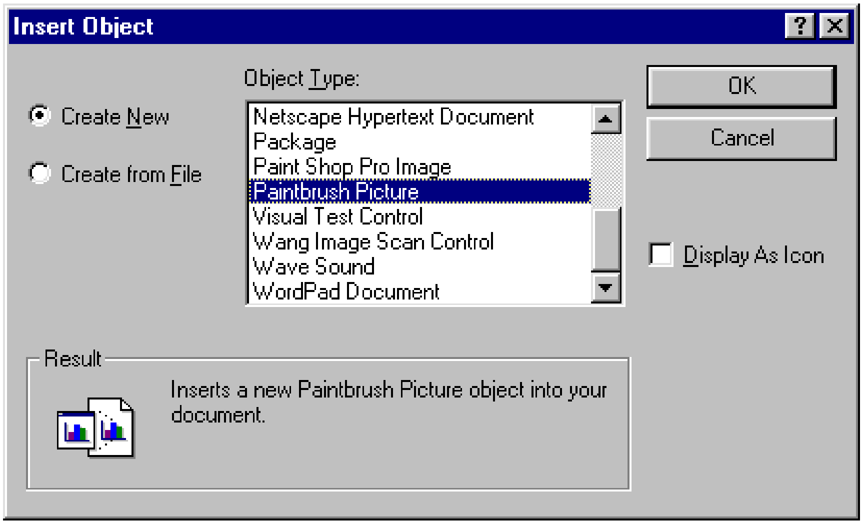

Figure 2.2 WordPad's Insert Object dialog.

Having chosen the Paintbrush picture object, the WordPad window shows a drawing region bounded on two sides by scroll bars. This is our embedded Paintbrush picture object. Notice too that the horizontal toolbar has gone away, to be replaced by a vertical toolbar and a pane at the bottom of the window containing a color palette. In addition, the menu bar has changed.

The new toolbar, pane, and menu bar are actually part of the user interface for the bundled Windows application Paint (formerly Paintbrush). We can confirm this by going to the **Help** menu; the "About" option is now **About Paint**.

The Paint toolbar, pane, and menus remain available as long as the picture object is *active*. The object is active when we can draw in the region; activation ends when we click outside the region or press Esc. Because the picture object is active inside the WordPad document, this sort of activation is known as *in-place activation*. The opposite of an in-place activation would be one where the picture object started up in a separate Paint window instead of in the container, WordPad.

If we click outside the drawing region, the activation is ended and the Paintbrush tools go away. The WordPad toolbar returns. However, the picture we have drawn remains embedded. We can save the document and, when we open it again in WordPad, the embedded Paintbrush picture will still be available for viewing and editing.

Thus we have created a compound document comprised of data from WordPad and Paint. The document remains principally a WordPad document—WordPad will be the application it will open in by default—but clearly it is not just a WordPad document.

We can continue working with this document to demonstrate the difference between linking and embedding, and between in-place and out-of-place activation. We will now link a file containing a picture object into the document.

Start Paint up from the **Start** menu with **Programs > Accessories > Paint**. Now draw something in the empty picture, and save it in a new file.

Return to the WordPad document. We are now going to link our new picture file into the WordPad document. Make sure the existing picture object is not activated, because we cannot link a picture object into a picture object. (Notice there is no **Insert > Object** command in WordPad while a picture object is activated—we can conclude from this that Paint can only play the role of a server in a compound document, and not that of a container.)

Choose **Insert > Object** in WordPad and select the **Create from File** option button. Select your Paint picture file in the **File** box.

If we clicked **OK** at this stage, the object in the Paint picture file would be embedded in the WordPad compound document. Future changes to the Paint picture file would not be reflected in WordPad document. However, if we select the **Link** box before clicking **OK**, the picture object would be stored as a link to the Paint picture file, so future changes to the file would be reflected in the WordPad document.

Select the **Link** box and click **OK**. If we double-click on the inserted picture to active it, the activation occurs in a new Paint window. Linked objects can never be edited in place. To end this sort of activation, we have to exit the Paint window.

2.2 Overview of OLE/COM and the Harlequin Dylan API

The aim of this section is to help readers who have never read about COM, OLE, and ActiveX before to understand them well enough to be able to use some of the higher-level support for these technologies that Harlequin Dylan provides. Accompanying notes explain Harlequin Dylan's approach to implementing these traditionally C-based technologies.

This is a large and complicated subject area and it is beyond the scope of this manual to describe it completely. You are likely to need more conceptual and technical information about these technologies. The Microsoft C API documentation for COM, OLE, and ActiveX will help; for a more detailed conceptual overview, we also recommend the book *Understanding ActiveX and OLE* by David Chappell (Microsoft Press, 1996; ISBN 1-57231-216-5).

2.2.1 COM interfaces and methods

In COM, software components communicate by means of *interfaces*. Interfaces are implemented by server components and used by client components. COM objects consist of one or more interfaces, usually two or more.

Interfaces consist of a number of *methods*. Methods are functions that client software can call to access the server's features. Note that these methods are not the same as Dylan methods: where necessary, we call them *COM methods* to distinguish them from Dylan methods.

Clients obtain pointers to interfaces in order to call the COM methods they contain. They do so by communicating with a COM support library. Microsoft Windows 95 and Windows NT both contain such a library. The Harlequin Dylan OLE/COM libraries make calls to this library as necessary.

COM defines a number of standard interfaces. The fundamental COM interface, which every COM object must support, is called **`IUnknown`**. (By convention, interface names always start with an 'I'.)

When a client instantiates a COM object, it typically gets back a pointer to **`IUnknown`**. After this, the client can attempt to get pointers to the other interfaces the object provides. For example, a word processor might implement an interface called **IContents** that provides services for creating and manipulating a table of contents, and an interface called **IPrint** that provides printing services.

IUnknown provides three methods: **QueryInterface**, **AddRef**, and **Release**. **QueryInterface** is used to get pointers to other interfaces supported by the COM object, while **AddRef** and **Release** increment and decrement a reference count to enable the server to know when all clients have finished using the interface.

All COM interfaces inherit from the **IUnknown** interface. This *does not* mean they inherit the implementation of the **IUnknown** methods, just that they inherit the interface itself, meaning that all COM interfaces are required to implement **QueryInterface**, **AddRef** and **Release** themselves. Thus COM has interface inheritance but not implementation inheritance.

In the C/C++ world, COM developers would therefore have to include code for the three **IUnknown** methods in every interface they write, and also to repeat the code of any COM methods their interface might inherit from other interfaces.

Harlequin Dylan represents interfaces using Dylan classes, and represents their methods using Dylan generic function methods specialized on those interface classes. All the Harlequin Dylan OLE/COM API libraries export the Dylan class **<IUnknown>**, which represents the basic COM interface **IUnknown**.

Because COM specifies that the **IUnknown** interface must support **QueryInterface**, **AddRef**, and **Release**, the Harlequin Dylan OLE/COM API libraries define Dylan methods for these operations on **<IUnknown>**.

Every COM interface in Dylan must be a subclass of **<IUnknown>**. This means that, through the standard Dylan generic function dispatch mechanisms, every COM interface you define will inherit the implementations of **QueryInterface**, **AddRef**, and **Release** that are defined on the Dylan class **<IUnknown>**. Thus COM's interface-inheritance-only limitation is overcome naturally.

If you prefer, you can override the inherited methods in the normal Dylan fashion, by adding Dylan methods specializing on your subclass of **<IUnknown>**.

2.2.2 COM objects and classes

COM software is implemented in terms of *component objects* or *objects*—hence the name “Component Object Model”. Every COM object is an instance of a *COM class* or *coclass*.

The terminology here is intentionally similar to that used in object-oriented programming languages: a coclass describes the complete set of properties and behavior in a software component, while a COM object is a run-time instance of a coclass. More than one instance of a particular coclass can exist at a time, just as you might have several instances of **`<integer>`** in a Dylan program.

For example, suppose you implemented an online help viewer as a coclass. The coclass would be implemented as part of a server component—an EXE, DLL, or OCX. Client applications that wanted to provide an online help system could do so by asking the server application to create an instance of the help viewer coclass. Once the help viewer object was created, the client could get hold of the appropriate interfaces and use them to send the help viewer some text to display. Multiple viewer objects of the same coclass could be instantiated if necessary.

2.2.3 Representing interfaces

COM represents interfaces using a simple table-and-pointer mechanism. When a client connects to an object and requests an interface, it receives a C pointer to the interface. The interface pointer leads to a table of pointers (called a *vtable*) each of which point to the COM methods of the interface.

Suppose there was an interface **`IPaint`** with the methods **`Draw`** and **`Fill`** in addition to the methods **`QueryInterface`**, **`AddRef`** and **`Release`** that are inherited from **`IUnknown`**. Figure 2.3 shows how a connection to **`IPaint`** would be represented.

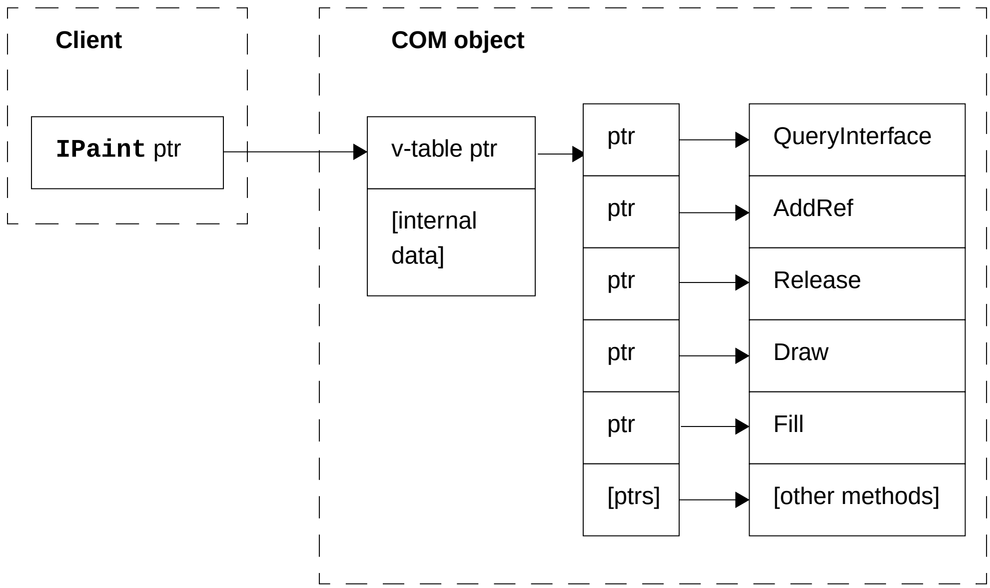

Figure 2.3 COM interface representation.

In an OLE/COM application built with Harlequin Dylan libraries, you often do not need to represent COM interfaces, classes, or methods explicitly. The libraries instead provide high-level protocols that implement the necessary COM details for you. You may be required to implement certain Dylan methods or extend certain Dylan classes, but usually do not need to represent a COM class, interface, or method directly. This is very convenient, and adequate for most applications.

2.2.4 Dispatch interfaces

The interface mechanism commonly used in OLE Automation deserves particular mention. As we have seen, server components can expose their services to clients through named interfaces: a simple painting server might offer a drawing interface called **`IPaint`**, while a word processor might offer access to its thesaurus through an interface called **`IThesaurus`**, and access to its indexing tools through an interface called **`IIndex`**.

But in Automation it is common to use a different means of exposing services to clients. Rather than defining named interfaces, Automation servers implement a *dispatch interface* or *dispinterface* protocol, where clients access COM methods through a dispatching mechanism instead of on a call-by-name basis.

The dispinterface protocol is defined by the standard COM interface **`IDispatch`**. This interface defines four COM methods: **`Invoke`**, **`GetTypeInfo`**, **`GetIDsOfNames`**, and **`GetTypeInfoCount`**. It also inherits from **`IUnknown`**, as all COM interfaces do, and so must provide implementations for the **`QueryInterface`**, **`AddRef`** and **`Release`** methods.

Every other COM method that a server defines for the dispinterface has its own identifier, called a *dispatch identifier* or *DISPID*. Rather than call these methods using a name, clients call **`Invoke`** and pass it the DISPID of the method they want to invoke.

As well as COM methods, dispinterfaces can also have *properties*. Properties are values that can be retrieved or set, just like slots in Dylan classes. Like the methods in a dispinterface, properties are accessed using **`IDispatch`**'s standard method **`Invoke`**.

Harlequin Dylan's high-level interface to Automation is the OLE-Automation library. This library exports the Dylan class **`<simple-dispatch>`**, which represents the COM interface **`IDispatch`**.

You can define dispinterfaces of your own by defining a subclass of **`<simple-dispatch>`**. You can then define dispatch methods for your dispinterface by writing Dylan methods on your subclass of **`<simple-dispatch>`**. Because **`<simple-dispatch>`** is a subclass of **`<IUnknown>`**, it inherits the implementations of the three **`IUnknown`** methods.

You can define properties for your dispinterface simply by adding slots to your subclass definition.

2.2.5 GUIDs: Globally unique identifiers

Globally unique identifiers (GUIDs) are a core concept in COM. Client components use them to identify the COM interfaces, objects, and, in dispinterfaces, methods, implemented by server components. Servers use them to register their services with Windows (or, more generally, with any computer system that supports COM) so that clients can find them.

GUIDs are generated in such a way that they are always globally unique. Thus, through GUIDs, COM ensures that server components can be identified uniquely on any computer system, anywhere.

When it comes to GUIDs, COM terminology can be confusing, because COM uses different abbreviations to refer to a GUID according to the context in which the GUID is being used. Here is a rough guide to the usage:

GUID	Globally unique identifier. All the identifier abbreviations in this list are merely aliases for "GUID".
CLSID	Class ID. A GUID used to identify a COM class.
UUID	Universally unique identifier. Another name for GUID.
IID	Interface ID. A GUID used to identify a COM interface.

Servers and clients do not generate GUIDs by themselves. You must supply the GUIDs and embed them—as constant values—in your software. GUIDs must remain fixed for all time so that clients will always be able to find your COM class or the interfaces it provides.

Once you have the GUIDs you need for the software you are writing, you can associate those GUIDs with its interfaces and COM classes. Thereafter, you can register your COM software, and the services it provides, on a machine that supports COM, and other COM software will be able to find and use your software using GUIDs to identify it. COM software is registered in the Windows Registry.

There are two stages to creating GUIDs for use in applications. First, you need to generate the ID number. Then the ID number has to be encoded in a special C data structure that the Windows OLE/COM libraries understand. The Harlequin Dylan OLE/COM API libraries need to pass GUIDs to Windows in this C format. See Section 2.5.1 on page 106 for details of how you can do both things.

2.2.6 Type information and type libraries

COM uses *type information* to make formal descriptions of COM objects and their interfaces. Type information encodes such details as the names of interfaces, the GUIDs that identify them, their methods, and their methods' argument names and types.

With type information, all the interfaces that a COM object implements, and all the methods and properties of those interfaces, can be described in a way that client software can understand. Clients can use type information to determine how to communicate with COM objects, and to make decisions about whether to use them.

Clients get hold of type information through a *type library*. A type library is a representation of the type information for a COM object that client software can query to find out about the object's interfaces.

How does type information become a type library? As a server's developer, you enter the COM type information as part of the server's object and interface definitions. Harlequin Dylan takes care of gathering the information into a type library, creating the library file, and installing it in the Windows Registry. This all happens as part of Harlequin Dylan's server self-registration facilities.

2.3 Registering OLE/COM software

All the different kinds of OLE/COM server software—compound document server applications, Automation server applications, and OLE controls—must be registered with Windows so that OLE/COM client applications can use them. Until a server or control has been registered, there is no way for a client to find it.

Servers and controls typically register themselves. Registration involves the server or control entering its own COM Class ID into the Windows Registry. If a server or control implements more than one COM object, it must register the Class ID for each one.

The Class ID is recorded alongside the pathname to the server executable, or control OCX or DLL file. When a client application asks to connect to a COM object with a particular Class ID, the native Windows OLE/COM libraries consult the Registry to find the path to the server or control that implements the COM object in question, using the Class ID as a key.

Conventionally, you register server applications by calling them with the command-line argument **`/RegServer`** or **`-RegServer`**. For example:

```
ms-dos> my-server-app.exe /RegServer
```

The application recognizes the registration request, registers itself, and exits without doing anything else. Applications can also unregister themselves when they receive the command-line argument **`/UnregServer`** or **`-UnregServer`**.

OLE controls register themselves dynamically when they are invoked. You supply registration data (including Class IDs) to a macro **`initialize-ole-control`** (see page 302), which prepares the DLL / OCX into which the control is built for dynamic self-registration when a container invokes it. You can also register a control explicitly with the command **`regsvr32`**:

```
ms-dos> regsvr32 my-control.ocx
```

Server applications and controls must of course contain code that can perform registration. The Harlequin Dylan OLE / COM API provides functions to help simplify the coding you need to do for self-registration.

Note: When you perform an explicit registration with either **`/RegServer`**, **`-RegServer`**, or **`regsvr32`**, the full pathname of the file is recorded in the registry. To move the file to a different folder, you must first unregister it, then move it, and then register it again at its new location.

2.4 About the example OLE/COM programs

Harlequin Dylan includes a number of OLE / COM example programs. This section is a guide to building, registering, and using those examples.

The examples are all available from the Examples subfolder of your Harlequin Dylan installation. The Harlequin Dylan IDE's Examples dialog also gives us a view of these examples as projects that we can build and (in some cases) test from the IDE. However, because we need to register those examples that implement servers and controls we do generally need to take the folder-level view of them rather than the abstract view supplied by the Examples dialog.

Under the Examples folder is the OLE folder that contains all the examples of interest to us here.

2.4.1 OLE examples

The OLE examples are in the OLE folder under the Examples subfolder to your top-level Harlequin Dylan installation folder:

Bank — A three-tier example application that illustrates the use of custom (dual) interfaces to provide the intrastructure of a banking application.

OCX-Scribble — An OLE control that implements a simple drawing application.

OLE-Scribble — An OLE compound document server that implements a simple drawing application.

The applications in OCX-Scribble and OLE-Scribble are functionally the same except that one is built as a control and the other as a compound document server. (Thus they must support different COM interfaces in order to function as a control or server.)

Button-OCX — An OLE control that uses a DUIM gadget to implement a simple button.

Sample-DUIM-Container — An OLE container application, which uses a DUIM framework for the user interface, that can embed graphical objects from other programs.

The examples listed below demonstrate the use of low-level OLE interfaces to support creating and embedding graphical objects.

Sample-OLE-Server — An OLE compound document server application that can create COM objects that can be embedded in a compound document container application.

Sample-OLE-Container — An OLE compound document container application that can embed COM objects created by compound document server applications.

2.4.2 Win32 API OLE examples

Under the OLE folder, there are folders each containing a different example with its GUI implemented directly through Win32 API calls:

Sample-Automation-Controller

An OLE Automation controller application that can send commands to the Sample-Automation-Server application.

Sample-Automation-Server

An OLE Automation server application that can act upon commands received from the Sample-Automation-Controller application.

Win32-OLE-Server

An OLE compound document server application that creates COM objects that can be embedded in a compound document container application.

Win32-Invisible-Control

An application demonstrating the use of the OLE-Control-Framework library to create an invisible OLE Control (an Active X Control).

The applications built in Sample-Automation-Controller and Sample-Automation-Server really comprise a single example, intended to work together. The buttons of the Automation controller application allow you to issue drawing commands to the Automation server, which draws circles and squares in pen colors you select from a color-chooser dialog.

2.4.3 Building and registering the examples

To build any of these examples, first open it from the Harlequin Dylan IDE Examples dialog or by opening its project (`.HDP`) file. Then build it by choosing **Application > Build** in the example's project window.

If the example is a server or a control, you will also need to register the file that was built from it. Each project has a **`README.html`** file that describes how to register it.

2.4.4 Running the examples

In each example project there is a **README.html** file that describes how to run the example.

2.5 Making GUIDs

There are two stages to making a GUID for use in an application. First you need to create the unique number that makes up the GUID, and then you need to create a Dylan object representing the GUID.

2.5.1 Creating GUID numbers

To create the numbers that make up a GUID, you need to run a special utility. Harlequin Dylan offers a utility called **create-id** for generating GUIDs. Owners of Microsoft Windows programming tools and development kits have a further choice with the utility UUIDGEN. (UUID is another name for GUID.) You can use whichever utility you like.

The **create-id** utility is a console-mode program that you run from an MS-DOS command prompt. You can find it in the Bin subfolder of your Harlequin Dylan installation.

The **create-id** utility accepts a single argument, which is the number of GUIDs you wish generate. If you do not supply an argument, the default is to generate a single GUID. The script writes the generated GUIDs to the standard output in the form of string literals that you can paste into Dylan source code.

This is an example of using **create-id** to create four GUID numbers:

```
ms-dos> create-id 4
"{C16C1BFE-41DA-11D1-9A58-006097C90313}"
"{C16C1BFF-41DA-11D1-9A58-006097C90313}"
"{C16C1C00-41DA-11D1-9A58-006097C90313}"
"{C16C1C01-41DA-11D1-9A58-006097C90313}"
ms-dos>
```

2.5.2 Making a GUID instance

After generating all the GUID numbers you need for the different parts of your application, the next stage is to supply those GUIDs in the proper format when you instantiate the classes, interfaces, and so on that the GUIDs will identify.

You need to supply GUIDs when making instances of classes such as **`<embeddable-frame>`** (from the DUIM-OLE-Server and DUIM-OLE-Control libraries), and**`<class-factory>`** (from the OLE-Automation and OLE-Server libraries).

The Harlequin Dylan OLE/COM API offers two ways of doing this. The simplest, but least efficient, method is to pass the GUID as a string, in the format used in the output of the **`create-id`** utility (see "Creating GUID numbers", above). That format is:

"{*xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*}"

Where each *x* is a hexadecimal digit. For example:

```
"{C16C1BFE-41DA-11D1-9A58-006097C90313}"
```

You could paste these strings directly into your code, to server as init arguments for the various classes that require GUIDs.

The second, and more efficient, way to supply a GUID is to pass the output from **`create-id`** as a series of arguments to the function **`make-GUID`**, which creates the C data that the native Windows OLE/COM libraries need to represent the GUID, and returns a handle on that data as an instance of the class **`<REFGUID>`**. (The class **`<REFGUID>`** also turns up in libraries under the aliases **`<REFCLSID>`** and **`<REFIID>`**.)

The DUIM-OLE-Server, DUIM-OLE-Control, OLE-Automation, OLE-Server and OLE-Control-Framework libraries all export **`make-GUID`**.

The following is a brief description of **`make-GUID`**. See page 313 for a full reference description.

make-GUID *Function*

`make-GUID` ***l w1 w2 b1 b2 b3 b4 b5 b6 b7 b8*** **=>** ***ID***

Returns an object representing a a globally unique identifier (GUID), created from the arguments. The ID value is an instance of the class **<REF-GUID>**. This function creates the C structure necessary to represent a GUID in the native Windows OLE/COM libraries.

The arguments to **make-GUID** are taken from the output of the GUID-generation utility **create-id**. Given the following value from **create-id**:

```
"{113f2c00-f87b-11cf-89fd-02070119f639}"
```

you need to split the value into parts as follows, and for each add the prefix #x. For example:

```
define constant $my-class-ID =
  make-GUID(#x113f2c00, #xf87b, #x11cf, #x89, #xfd, #x02, #x07,
            #x01, #x19, #xf6, #x39);
```

2.6 Supported interfaces

The following is a list of COM interfaces the Dylan OLE, COM, and ActiveX libraries implement.

- The COM library implements IUnknown and IClassFactory.
- The OLE-Automation library implements ITypeInfo and IDispatch.
- The OLE-Server library implements IDataObject, IEnumFORMATETC, IEXternalConnection, IOleInPlaceActiveObject, IOleInPlaceObject, IOleObject, IPersistStorage, and IViewObject2.
- The OLE-Control-Framework library implements IOleControl, IPersistStreamInit, IProvideClassInfo, and IProvideClassInfo2.

2.7 Libraries in the Harlequin Dylan OLE/COM API

Harlequin Dylan provides access to OLE/COM facilities through libraries at three levels of abstraction. The following sections discuss each level.

2.7.1 High-level DUIM integration

At the highest level, you can write slightly modified versions of normal DUIM applications that will function as OLE servers and controls. The DUIM-OLE-Server and DUIM-OLE-Control libraries provide a simple framework for building OLE server applications and OLE controls from DUIM applications. These libraries integrate smoothly with the DUIM programming model. See Chapter 3, "Compound Documents and OLE Controls in DUIM" for details of these libraries.

There is presently no high-level framework for developing DUIM applications that can be used as OLE containers.

2.7.2 Low-level FFI libraries

At the lowest level, the COM, OLE, OLE-Dialogs, OLE-Controls, and OLE-Automation libraries present a simple mapping of Dylan names to names defined in the Microsoft OLE / COM API. With these libraries you can build OLE / COM components without using DUIM.

We built these libraries using the Harlequin Dylan C foreign function interface library (C-FFI). Although they are low-level, note that the libraries do make all the conversions necessary to pass Dylan data to C functions and to convert C return values into Dylan objects.

Dylan library	C header	Link library	Runtime library
COM	OBJBASE.H	OLE32.LIB,UUID.LIB	OLE32.DLL
OLE	OLE2.H	OLE32.LIB,UUID.LIB	OLE32.DLL
OLE-Automation	OLEAUTO.H	OLEAUT32.LIB	OLEAUT32.DLL
OLE-Dialogs	OLEDLG.H	OLEDLG.LIB	OLEDLG.DLL
OLE-Controls	OLECTL.H	OLEPRO32.LIB	OLEPRO32.DLL

Table 2.1 OLE/COM FFI libraries and the corresponding Windows files.

These FFI libraries enable Dylan programs to use OLE / COM in much the same way as a C++ program would, assuming it used direct OLE calls rather than MFC. You can use most of function, type, variable, and constant names documented in the OLE specifications, though of course there are a few syntactic modifications to account for Dylan naming and coding conventions. Chapter 9, “OLE FFI Facilities”, describes these differences.

2.7.3 Intermediate layer libraries

Between the high-level DUIM frameworks and the low-level FFI layer is an intermediate layer of OLE / COM libraries. The OLE-Server and OLE-Control-Framework libraries support compound documents and OLE controls respectively; they are comparable to DUIM-OLE-Server and DUIM-OLE-Control, but for use when building application user interfaces with the direct Win32 API libraries (Win32-User and Win32-GDI) instead of DUIM. The OLE-Automation library includes a high-level framework for OLE Automation as well as a low-level FFI interface to the Microsoft OLE Automation API, as exposed through **OLEAUTO.H**, **OLEAUT32.LIB**, and **OLEAUT32.DLL**.

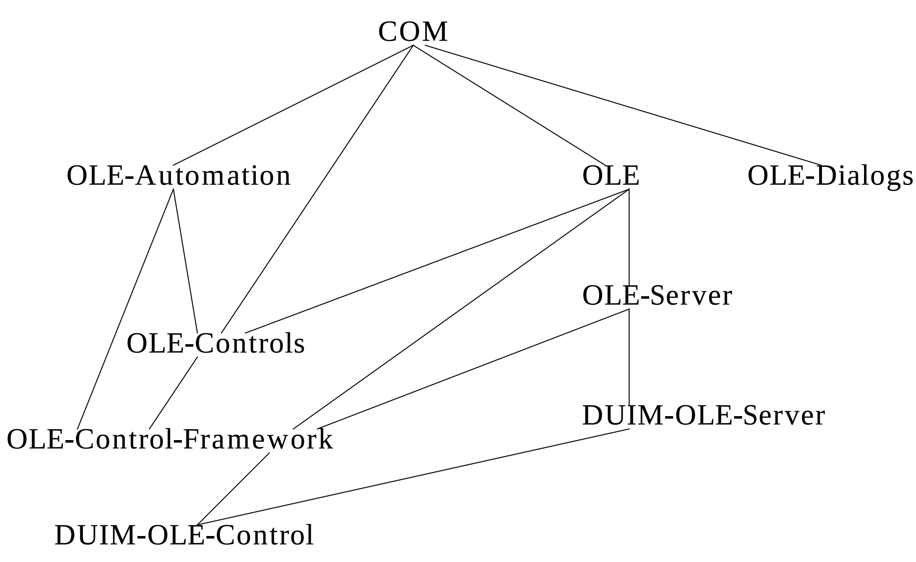

Figure 2.4 The Harlequin Dylan OLE/COM libraries.

These libraries are covered in successive chapters:

- Chapter 4, "OLE Automation" for details of the OLE-Automation library.
- Chapter 6, "The OLE-Server Library".
- Chapter 7, "The OLE-Control-Framework Library".

2.7.4 Basic COM integration and utilities

See Chapter 8, "Basic COM Integration", for details of basic COM integration features and protocols available through all OLE/COM libraries.

2.8 Choosing the right OLE/COM libraries

There is more than one way to write an OLE/COM component with Harlequin Dylan libraries. As we have seen, there are both high-level frameworks for developing particular kinds of components, and low-level FFI interfaces to the major Microsoft OLE/COM API libraries. Which Dylan library should you use to get the job done?

Most OLE/COM functionality is available through the low-level FFIs: OLE, COM, OLE-Dialogs, OLE-Control, and parts of OLE-Automation. You could use these libraries alone and achieve your goals, but it would be painful and unnecessary given that Harlequin Dylan also offers some higher-level abstractions. Obviously it is much better to start with the higher-level libraries, and delve into the lower levels only if you have certain requirements they do not satisfy—this is particularly true if you are not experienced in implementing OLE/COM-based software using the C-based Microsoft APIs.

The following sections explain which OLE/COM libraries are suitable for particular tasks.

2.8.1 For compound document server applications

The DUIM-OLE-Server and OLE-Server libraries are both high-level frameworks for developing compound document server applications. Your choice of library depends on whether you are using DUIM or the low-level Win32

FFI libraries (Win32-user and Win32-GDI; see the chapter on Win32 API libraries in the Interoperability I reference volume) to develop your server application's user interface.

The OLE library, an FFI interface to the C-based Win32 OLE library, is low-level but complete, and may be a useful supplement to the higher-level libraries.

DUIM-OLE-Server exports the module **`DUIM-OLE-Server`**, and OLE-Server exports the module **`OLE-Server`**.

The OLE library exports the module **`OLE`**.

2.8.2 For compound document container applications

We do not encourage writing compound document container applications with these libraries. There is presently no framework for this, whether written using DUIM or using the low-level Win32 FFI libraries.

2.8.3 For OLE controls

The DUIM-OLE-Control and OLE-Control-Framework libraries are both high-level frameworks for developing OLE controls. The choice of library depends on whether you are using DUIM or the low-level Win32 FFI libraries (Win32-user and Win32-GDI; see the chapter on Win32 API libraries in the Interoperability I reference volume) to develop your control's user interface.

The OLE and OLE-Controls libraries, both FFI interfaces to the C-based Win32 OLE (compound documents) and OLEPRO32 (OLE controls) libraries, are low-level but complete, and may be a useful supplements to the higher-level libraries.

The DUIM-OLE-Control library exports the module **`DUIM-OLE-Control`**, and the OLE-Control-Framework library exports the module **`OLE-Control-Framework`**.

The OLE library exports the module **`OLE`**, and the OLE-Controls library exports the module **`OLE-Controls`**.

2.8.4 For OLE control container applications

We do not encourage writing OLE control container applications with these libraries. There is presently no high-level framework for developing control containers, whether written using DUIM or the low-level Win32 FFI libraries.

2.8.5 For OLE Automation servers and controllers

The OLE Automation library provides a high-level framework suitable for all OLE Automation implementations, and additionally provides an FFI to the Win32 OLEAUT32 library. You can use this library to implement OLE Automation applications regardless of whether you are using DUIM or the Win32 API to write your user interface. Indeed, you can use OLE Automation in console applications.

The OLE-Automation library exports the module **`OLE-Automation`**.

2.8.6 For OLE dialogs

The OLE-Dialogs library provides a low-level FFI interface to the OLE common dialogs of the C-based Win32 OLE-DLG library. The library provides the following dialogs: Insert Object dialog, Convert Object dialog, Paste Special dialog, Change Icon dialog, Edit Links dialog, Update Links dialog, Change Source dialog, Busy dialog, User Error Message dialog, and Object Properties dialog.

The OLE-Dialogs library exports the module **`OLE-Dialogs`**.

3

Compound Documents and OLE Controls in DUIM

3.1 Introduction

This chapter explains how to write DUIM applications that implement OLE compound document servers, by using the DUIM-OLE-Server library; and applications that implement OLE controls, by using the DUIM-OLE-Control library.

3.2 About the DUIM-OLE-Server library

The DUIM-OLE-Server library allows you to create a DUIM application that can be used as an OLE compound document server.

In other words, the application can either be run by itself, or it can be embedded within any OLE container application, such as WordPad. The DUIM-OLE-Server library is closely integrated with the normal DUIM programming model, so you do not need to know anything about the low-level Harlequin Dylan OLE/COM API or Microsoft's OLE/COM API to use it.

If you want to write a compound document server using Harlequin Dylan's direct Win32 API instead of DUIM, you should use the OLE-Server library. See "The OLE-Server Library" on page 237.

3.2.1 Limitations of the DUIM-OLE-Server library

There are some limitations to what you can do with the DUIM-OLE-Server library. These limitations may not prevent you from using standard OLE/COM features in your applications, but you will need to use the lower-level OLE-Server and OLE libraries to access the missing features. Note that the lower libraries are more difficult to use, and we make no guarantees that these limitations can be overcome. The limitations are:

- Cannot add more control or label panes around an in-place activation.
- No clipboard.
- No drag and drop.
- No context-sensitive help.
- No undo facility.
- No presentation as icon or thumbnail.
- Additional considerations for MDI context.
- Cloning of an embedded object.
- Using container's recommended color palette.

The library only allows you build an local compound document server — that is, an executable (**`.EXE`**) file with either in-place or out-of-place activation. The servers support object embedding only — not linking, where the server's persistent data is stored in its own file instead of as part of the compound document, with the compound document containing a link to the other file.

The library does not allow you to build an in-process server — that is, a server built as .DLL file that runs in the container's process.

3.2.2 Modules exported from DUIM-OLE-Server

The DUIM-OLE-Server library exports the module **`DUIM-OLE-Server`**. It also uses and re-exports all names from the OLE-Server library's **`OLE-Server`** module.

3.2.3 A note about compound document container applications and DUIM

We do not encourage writing compound document *container* applications with the present Harlequin Dylan OLE/COM API libraries, whether or not they are written in DUIM. You are free to try, but we cannot guarantee success.

3.3 About the DUIM-OLE-Control library

The DUIM-OLE-Control library allows you to build an OLE Control using DUIM to define its user interface elements. Using DUIM-OLE-Control is similar to using DUIM-OLE-Server; this reflects the fact that OLE Controls share some of the characteristics of OLE compound document servers, such as being embeddable.

Because DUIM-OLE-Control is integrated tightly with the standard DUIM programming model, you do not need to know anything about the low-level Harlequin Dylan OLE/COM API or Microsoft's OLE/COM API to use it.

If you want to write an OLE control using Harlequin Dylan's direct Win32 API instead of DUIM, you should use the OLE-Control-Framework library.

3.3.1 Modules exported from DUIM-OLE-Control

The DUIM-OLE-Control library exports the module **`DUIM-OLE-Control`**. It also uses and re-exports all names from the OLE-Control-Framework library's **`OLE-Control-Framework`** module.

3.3.2 A note about OLE control container applications and DUIM

We do not encourage writing OLE control *container* applications with the present Harlequin Dylan OLE/COM API libraries, whether or not they are written in DUIM. You are free to try, but we cannot guarantee success.

3.4 Building a compound document server in DUIM

This section gives an overview of building compound document servers in DUIM using the DUIM-OLE-Server library.

3.4.1 Differences from ordinary DUIM applications

To build a compound document server application in DUIM, write your application as you would a normal DUIM application, but with the few differences summarized below.

1. Use the DUIM-OLE-Server library.

   In addition to the DUIM libraries you would normally use, use the DUIM-OLE-Server library and its **`DUIM-OLE-Server`** module in your server application's library and module definitions.

2. Define the application as a subclass of **`<embeddable-frame>`**.

   Instead of defining your application as a subclass of **`<simple-frame>`**, define it as a subclass of **`<embeddable-frame>`** (page 124).

3. Give the application a COM Class ID (a GUID).

   When creating an instance of **`<embeddable-frame>`** with **`make`**, supply a **`class-id:`** argument (whose value is usually created by **`make-GUID`**, page 313). This argument is recorded next to the name of the server application in the Windows registry; it is the key that client applications use to find the server application. There are other, optional arguments that allow further control over registration. See **`<embeddable-frame>`**, page 124.

4. Define methods to handle persistent storage in compound documents.

   Define methods for **`save-frame-to-storage`** (page 136) and **`load-frame-from-storage`** (page 137). These functions handle your server application's persistent data, ensuring that it can be saved as part of a compound document and reloaded next time the user accesses the compound document. When your application needs to save its data, call **`note-embedded-data-changed`** (page 132) on the instance of **`<embeddable-frame>`**.

5. Use **`frame-status-message`** for application status bar messages

   To display status bar messages, instead of referring directly to **`frame-status-bar`**, use this idiom:

```
frame-status-message(frame) := "Message-text string";
```

This ensures that status bar messages always work, whether or not the application is running embedded.

For more information, see the code for the DUIM OLE Scribble example in:

```
Examples\duim-ole\ole-scribble\
```

3.4.2 Running your DUIM-OLE-Server application

You run DUIM-OLE-Server applications by calling **start-frame** on the application frame, as you would any DUIM application. Note that there is a specialized method on **start-frame** for **<embeddable-frame>**, so do not try to use the frame in some other way.

3.4.3 Registering server applications with Windows

Before your application can be embedded in a container, it must first be registered with the Windows operating system as an OLE server. Once you have done this, container applications can use your application as a server.

The typical way to register a server application is to invoke it from the MS-DOS command line, passing the argument **/RegServer**. The application recognizes the registration request, creates the necessary entries in the Windows registry, and then exits (without displaying any window). You can revoke the registration — that is, remove the registry entries — by running the application again with the option **/UnregServer**. If you do not specify either argument, the application runs normally.

The DUIM-OLE-Server library takes care of these registration tasks within the normal DUIM programming context. You do not have to write any code for interpreting the command-line arguments. Instead, when you call **start-frame** on your instance of **<embeddable-frame>**, the selected method checks whether you passed **/RegServer** was passed. If you did, the **start-frame** method registers your server application and then exits the application.

For example:

```
ms-dos> my-server-app.exe /RegServer
```

Note: The full pathname of the executable file is recorded in the registry. If you move the file to different folder afterwards, you must register it again.

3.4.4 Specifying menus for in-place activation

When a server application is activated in-place (that is, in the container's window), OLE merges its frame's menu bar into the container application's menu bar. Often, it is more appropriate to have a different set of menus available during the server's in-place activation. You can add a method to **frame-container-menus** to specify different menus to be seen when your application is running in-place.

If the application is run out of place (that is, in its own window), it uses **frame-menu-bar** and never calls **frame-container-menus**.

3.4.5 Skeleton application

The overall structure of an OLE server program should look something like the skeleton application below.

```
define frame <foo-frame> (<embeddable-frame>) ... end;

define constant $foo-class-ID = make-GUID(...);

define method save-frame-to-storage ... end;

define method load-frame-from-storage ... end;

define variable $foo-ole-server
  = make(<foo-frame>,
         title: "Foo Works",
         class-id: $foo-class-ID,
         prog-id: "DUIMSVR.TEST.1",
         object-title: "FooSoft FOO");

start-frame($foo-ole-server)
```

Here we have a server application that implements "foo" objects that can be embedded in a compound document. The **title:** init-keyword names the server application window, and the **object-title:** init-keyword is the name that appears in the container's Insert Object dialog. The **class-id:** is the

GUID that will be entered into the registry for identifying the COM object that the server implements, and the **prog-id:** is the name associated with the server in the registry.

3.4.6 Example DUIM-OLE-Server application

You can find a complete example in the following Harlequin Dylan installation folder:

```
Examples\duim-ole\ole-scribble\
```

See its **README.html** file for a discussion.

3.5 Building an OLE control in DUIM

This section gives an overview of building OLE controls in DUIM using the DUIM-OLE-Control library. It assumes that you have already read about implementing OLE compound document servers in Section 3.4 on page 117.

3.5.1 Differences from implementing OLE compound document servers

To build an OLE control in DUIM using the DUIM-OLE-Control library, write your application as you would write an OLE compound document server with DUIM-OLE-Server (see Section 3.4), but with the few differences summarized below.

1. Use the library DUIM-OLE-Control instead of DUIM-OLE-Server.

   In addition to the DUIM libraries you would normally use, use the DUIM-OLE-Control library and its **DUIM-OLE-Control** module in your control's library and module definitions.

2. You can optionally provide type information, specifying the methods and properties that are available to your control's clients (control containers). Type information is represented with an instance of **<coclass-type-info>**. If the frame just contains a DUIM **<gadget>**, you can let the library create type information automatically. It creates type information corresponding to the DUIM gadget protocols.

3. If possible, define a method for **max-storage-size**.

4. The program must be linked as a dynamic link library (file suffix .DLL or .OCX), not as an .EXE file.
5. Do not directly instantiate the DUIM frame or call **start-frame**.

   OLE control registration and activation is handled by the **initialize-ole-control** macro, described below.

An OLE control also differs from a compound document server in that out-of-place activation is never used, and to be actually used as a “control” element, the frame will typically contain just a single **<gadget>**, with no menu bar or tool bar, so that it can be active at the same time as other controls.

For more information, see the code for the DUIM OCX Scribble example, in

```
Examples\duim-ole\ocx-scribble\
```

3.5.2 Getting the control container’s ambient properties

Your OLE control can adapt itself to match different control containers by accessing and then acting upon the control container’s ambient properties.

To access the ambient properties of the container using it, your OLE control should call **frame-ole-object** on its own frame, and then use the facilities described in the documentation for the OLE-Control-Framework library, in “Ambient properties” on page 299.

The following example shows how a control can effectively localize itself according to the control container using it.

```
let obj = frame-ole-object(frame);
let locale-code = OLE-util-locale(obj);
let language = PRIMARYLANGID(locale-code);
select (language)
  $LANG-NEUTRAL,
  $LANG-ENGLISH => …
  $LANG-FRENCH  => …
  $LANG-SPANISH => …
…
```

3.5.3 Skeleton application

A DUIM OLE control program should look something like the skeleton application below.

```
define frame <my-frame> (<embeddable-frame>) ... end;

define coclass my-type-info
  uuid "{...}";
  ...
end;

initialize-ole-control(typeinfo: my-type-info,
                       prog-id: "my.prog.id",
                       title: "my control",
                       frame-class: <my-frame>);
```

The **coclass** definition should include at least one interface. See "Overview of OLE Automation" on page 148.

The **coclass** definition should specify the component-class option as either **<ocx-dispatch>** or a user-defined subclass thereof. The methods for automation properties and methods will be specialized on that class, and should use the function **ocx-frame** to find the corresponding frame object.

3.5.4 Example DUIM-OLE-Control application

You can find a complete example in the following Harlequin Dylan installation folder:

```
Examples\duim-ole\ocx-scribble\
```

See its **README.html** file for a discussion.

Harlequin Dylan does not include a control container for testing this example, but if you have Microsoft Visual C++ there is a Test Container application you can use.

You can also embed the control in a compound document container, such as WordPad, but its properties will not be accessible. To do this, first build the control in the Harlequin Dylan IDE, then register the control with **regsvr32 ocx-scribble.dll** at an MS-DOS prompt, and then use **Insert > Object** and choose **DUIM OCX Scribble** from the Object Type list.

3.6 The DUIM-OLE-SERVER module

This section contains a reference entry for each name exported by the DUIM-OLE-Server library's **DUIM-OLE-Server** module.

DUIM-OLE-Server also re-exports all the items from the OLE-Server library. See "The OLE-SERVER module" on page 243 for those reference entries.

<embeddable-frame> *Open abstract class*

Summary The class of DUIM applications that can be used as OLE compound document servers.

Superclasses **`<simple-frame>`**

Init-keywords

`class-id:` The COM Class ID that identifies the object the server provides. Required. This ID can be represented either as a string of 32 hexadecimal digits in the following format

"{*xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*}"

That is, where each *x* is a hexadecimal digit. For example:

"{e90f09e0-43db-11d0-8a04-02070119f639}"

Alternatively, the ID can be a **`<REFCLSID>`**, as returned from **`make-GUID`**.

You can get the ID value by generating a GUID with Harlequin Dylan's **`create-id`** utility. See "Creating GUID numbers" on page 106.

`object-title:` If the container application has an Insert Object dialog, this optional string argument will be used as the title of the COM object that your application serves. If this argument is not provided, the inherited DUIM frame **`title:`** is used instead.

At least one of **`title:`** and **`object-title:`** must be supplied for the application to be able to register itself. The object title should be unique, since if two servers are registered with the same title, only one will be accessible from the Insert Object dialog box.

short-title: A string used as the program name in container menus and the Links dialog box. It must not be more than 15 characters long. If not specified, it defaults to the shorter of **title:** or **object-title:**, truncated to 15 characters if necessary.

prog-id: The server application's programmatic identifier (prog ID). The prog ID is a string which is used internally and is only visible in the Registry Editor. If specified, it must start with a letter; it cannot contain any spaces or punctuation except period; and it must not be more than 39 characters long. It must be different from the ID of any other application. If not specified, a default value is created automatically using portions of the title and class ID.

Description

The class of DUIM applications that can be used as OLE compound document servers. Use it as the superclass of your DUIM application.

The three optional keyword parameters are used only during self-registration. You can override them by entering different values into the Windows registry by hand. If you need more control over registration, write override methods on the generic functions **frame-register-server** and **frame-unregister-server**; each takes the frame instance as its single argument.

None of this information is used to register OLE Controls written with the DUIM-OLE-Control library, since registration and instantiation is not controlled by the frame.

frame-container-menus *Open generic function*

Summary
Specifies menus that should be included in the container application's menu bar if this server application is activated in place.

Signature
frame-container-menus ***frame*** **=>** ***edit-menus object-menus help-menus***

Arguments
frame An instance of **<embeddable-frame>**.

Values
edit-menus See Description.

object-menus See Description.

help-menus See Description.

Description
Specifies menus to be included in the container's menu bar if the server application is activated in place.

Applications can optionally define a method on this function, specialized on the application's subclass of **<embeddable-frame>**. The method should return three values, corresponding to three positions in the menu bar that are designated for edit menus, object menus, and help menus respectively. Each of the three values can be any of the following:

#f, indicating that there are no menus in this group.

A menu object (an instance of **<menu>**).

A **<sequence>** of **<menu>** objects, possibly empty.

A menu bar object (an instance of **<sheet>**).

The default method returns:

```
values(frame-menu-bar(frame), #f, #f)
```

frame-container-name *Function*

Summary
: If the application is running as an OLE server, this function returns the name of the container application and the name of the container document.

Signature
: **`frame-container-name`** ***frame*** **`=>`** ***container document***

Arguments
: *frame* An instance of **`<embeddable-frame>`**.

Values
: *container* An instance of **`false-or(<string>)`**.

document An instance of **`false-or(<string>)`**.

Description
: Returns the name of the container application and the name of the container document.

Both values are instances of **`false-or(<string>)`**. The value *container* is **`#f`** if the application is not running as an OLE server; the value *document* is **`#f`** if there is no applicable container document name.

This function could return false if it is called too early during the server activation process, before the container has provided the information necessary.

An application running in an out-of-place activation can use this function to set the **Exit** menu item to **Exit to** *container*.

frame-embedded-sheet *Open generic function*

Summary
: Returns an instance of **`<sheet>`** representing the image that should be displayed in the container application.

Signature
: **`frame-embedded-sheet`** ***frame*** **`=>`** ***sheet***

Arguments
: *frame* An instance of **`<embeddable-frame>`**.

Values	*sheet*	An instance of **`<sheet>`**.

Description

Returns an instance of **`<sheet>`** representing the image that should be displayed in the container application.

If your server application frame contains more than a single drawing pane, you should write a method on this function, specialized on the application's subclass of **`<embeddable-frame>`**, to return such a sheet. It must be a mirrored sheet, not just a layout.

When an out-of-place activation is completed, a DUIM **`<window-repaint-event>`** is invoked on *sheet*, and whatever it draws is what will appear in the container application.

frame-embedded-size *Function*

Summary

Returns the dimensions (in pixels) of the space currently reserved in the container application for the embedded server image.

Signature

`frame-embedded-size` ***frame*** **`=>`** ***width height***

Arguments	*frame*	An instance of **`<embeddable-frame>`**.
Values	*width*	An instance of **`<integer>`**.
	height	An instance of **`<integer>`**.

Description

Returns the dimensions (in pixels) of the space currently reserved in the container application for the embedded server image.

width and *height* are the same values as reported by the function **`note-embedded-region-changed`**.

frame-embedded-size-requested *Open generic function*

Summary Returns the dimensions (in pixels) of the space that should be reserved in the container application for the embedded server image.

Signature **`frame-embedded-size-requested`** ***frame*** **`=>`** ***width height***

Arguments *frame* An instance of **`<embeddable-frame>`**.

Values *width* An instance of **`<integer>`**.

height An instance of **`<integer>`**.

Description Returns the dimensions (in pixels) of the space that should be reserved in the container application for the embedded server image.

You can optionally define a method on this function, specialized on your server application's subclass of **`<embeddable-frame>`**, to return the amount of space (in pixels) that you want to be reserved in the container for the embedded server image. The default method returns the size of the sheet returned by **`frame-embedded-sheet`**.

frame-init-new-object *Open generic function*

Summary An initialization hook for server applications.

Signature **`frame-init-new-object`** ***frame*** **`=> ()`**

Arguments *frame* An instance of **`<embeddable-frame>`**.

Values None.

Description An initialization hook for server applications.

When the low-level OLE / COM libraries are using your server application to create a new COM object (and not loading an old object with **`load-frame-from-storage`**), they call this generic function. You can define a method on this function to perform any initialization your server application might need. The default method does nothing.

frame-in-place-active? *Function*

Summary
: Returns true if the server application is running as an in-place activation, and false otherwise.

Signature
: **`frame-in-place-active?`** ***frame*** **`=>`** ***active?***

Arguments
: *frame* — An instance of **`<embeddable-frame>`**.

Values
: *active?* — An instance of **`<boolean>`**.

Description
: Returns true if the server application is running as an in-place activation, and false otherwise. Call this function to determine whether your application is currently running in place.

frame-ole-object *Function*

Summary
: Returns the OLE / COM interface object that the container application is using to access the server application.

Signature
: **`frame-ole-object`** ***frame*** **`=>`** ***obj***

Arguments
: *frame* — An instance of **`<embeddable-frame>`**.

Values
: *obj* — An instance of **`false-or(<DUIM-OLE-server>)`**.

Description Returns the OLE / COM interface object that the container application is using to access the server application.

The object, *obj*, is an instance of **false-or(<DUIM-OLE-server>)**. This function returns **#f** if your application is not currently connected to an OLE container.

The **<DUIM-OLE-Server>** class (from DUIM-OLE-Server) is a subclass of **<ole-server-framework>** (from the OLE-Server library) and of **<IUnknown>** (from the low-level COM library).

The interface object *obj* is the "controlling unknown" for all of the low-level OLE interfaces, so you can call **QueryInterface** on it if you need to obtain any of the low-level interface pointers. This function does *not* call **AddRef**, so a caller function should not call **Release** unless it also calls **AddRef** explicitly.

frame-ole-object-class *Open generic function*

Summary Returns the class to be instantiated for the OLE interface object, and a sequence of any additional keyword options to be passed to **make**.

Signature **frame-ole-object-class** ***frame*** **=>** ***class args***

Arguments *frame* An instance of **<embeddable-frame>**.

Values *class* An instance of **<class>**.

args An instance of **<sequence>**.

Description Returns the class to be instantiated for the OLE interface object, and a sequence of any additional keyword options to be passed to **make**. The *class* value is an instance of **<class>** and *args* an instance of **<sequence>**.

This function is not used for OLE controls.

The default method returns **`<DUIM-OLE-server>`** in *class*. The server application can define a method on this function that returns its own customized subclass. The subclass could do things like provide overrides for methods in either the DUIM-OLE-Server or OLE-Server libraries, or its **`initialize`** method could instantiate additional OLE interfaces (using the **`controlling-unknown:`** option to reference the object being instantiated) which would then be available to the container application when it calls **`QueryInterface`**. Note that an override method should concatenate the subclass's arguments with the second value returned by **`next-method()`**.

note-embedded-data-changed *Function*

Summary A server application should call this function on its frame object whenever a change is made to either the embedded object's persistent data or to the image that should appear in the container.

Signature **`note-embedded-data-changed`** ***frame*** **`=> ()`**

Arguments *frame* An instance of **`<embeddable-frame>`**.

Values None.

Description A server application should call this function on its frame object whenever a change is made to either the embedded object's persistent data or to the image that should appear in the container. The function ensures that **`save-frame-to-storage`** is called as necessary and that the container's display of the embedded object is refreshed as necessary. It has no effect if the program is not running under OLE.

This function is implemented as follows:

```
define method note-embedded-data-changed (frame ::
  <embeddable-frame>) => ()
    frame.embedded-data-changed? := #t;
      // need to save data

    note-embedded-image-changed(frame);
      // update image in container
end method;
```

If you prefer, your application can use the two elements of this function separately. The slot **embedded-data-changed?** indicates whether **save-frame-to-storage** needs to be called, and is automatically set to **#f** when either it or **load-frame-from-storage** is called.

note-embedded-region-changed *Open generic function*

Summary Notifies a server that the client has allocated the server a screen region different to the one it asked for.

Signature **note-embedded-region-changed** ***frame width height*** **=>** ***ok?***

Arguments *frame* An instance of **<embeddable-frame>**.

width An instance of **<integer>**.

height An instance of **<integer>**.

Values *ok?* An instance of **<boolean>**.

Description A server application can define a method on this function in order to be notified if the container application allocates a screen region different from that your server application asked for. For example, the container might permit the user to manually resize the embedded window. The *width* and *height* are given in pixels.

The default method does nothing; the embedded image is simply clipped if the server application tries to draw outside the provided space. Your application should provide a method if it wants to do something such as scaling the image to fit the available space. The method should return **`#t`** if the change is acceptable, or **`#f`** to force use of out-of-place activation if the server application cannot operate in the space given.

Note that the drawing pane sheet will be automatically resized as necessary independently of this function, but the server application should not rely on looking at the sheet size because it might reflect temporary visibility clipping, not necessarily the full image size.

Note that the same width and height values can also be obtained by calling **`frame-embedded-size`**, page 128, if you do not want to have to remember them here.

note-in-place-activation *Open generic function*

Summary A hook that the DUIM-OLE-Server library calls when it has activated the server application in place.

Signature **`note-in-place-activation`** ***frame*** **`=> ()`**

Arguments *frame* An instance of **`<embeddable-frame>`**.

Values None.

Description A hook that the DUIM-OLE-Server library calls when it has activated the server application in place. The call occurs immediately after the library has finished displaying the server application's document window, menus, and tool bar in the container application.

Your server application can define a method on this function if it wants to be notified of this event. The default method does nothing.

note-in-place-deactivation *Open generic function*

Summary
A hook that the DUIM-OLE-Server library calls just before it terminates the in-place activation of the server application.

Signature
`note-in-place-deactivation` ***frame*** **`=> ()`**

Arguments
frame An instance of **`<embeddable-frame>`**.

Values
None.

Description
A hook that the DUIM-OLE-Server library calls just before it terminates the in-place activation of the server application.

Your server application can define a method on this function if it wants to be notified of this event. The default method does nothing.

The library clears the container application's status bar before it calls this function, but you can call **`frame-status-message-setter`** here to leave a parting message.

save-frame-to-storage *Open generic function*

Summary
The DUIM-OLE-Server library calls this function to store the server application's data persistently as part of the container's compound document.

Signature
`save-frame-to-storage` ***frame stream*** **`=> ()`**

Arguments
frame An instance of **`<embeddable-frame>`**.

stream An instance of **`<storage-istream>`**.

Values
None.

Description The DUIM-OLE-Server library calls this function to store the server application's data persistently as part of the container's compound document. If your server application has any data that should be saved persistently, you must provide a method on this function. The default method does nothing.

If you wanted to save three **<integer>** values in the compound document, the method on **save-frame-to-storage** might look something like this:

```
define method save-frame-to-storage
  (frame :: <foo-frame>,
   stream :: <storage-istream>) => ()
     istream-write-integer(stream, frame.foo);
    istream-write-integer(stream, frame.bar);
    istream-write-integer(stream, frame.baz);
end method;
```

The function **istream-write-integer** stores a 32-bit signed integer (and returns no values), while **istream-write-int16** can be used for values that fit into a 16-bit signed field. Use **istream-write-float** for a **<single-float>**, or **istream-write-string** to write a **<byte-string>**.

You can also use the low-level OLE API function **IStream/Write** to output arbitrary FFI data, or you can use the functions in the Streams or Format modules of the IO library.Note that there are separate **istream-write-**… functions instead of a single generic function, because the object type is not saved to the stream; you need to know what kind of object you are writing in order to know how to read it back.

load-frame-from-storage *Open generic function*

Summary Restores data saved by **save-frame-to-storage**.

Signature **load-frame-from-storage** ***frame stream*** **=> ()**

Arguments	*frame*	An instance of **<embeddable-frame>**.
	stream	An instance of **<storage-istream>**.

Values None.

Description Restores data saved by **save-frame-to-storage**. Server applications must provide a method on this function. The default method does nothing.

Server applications do not call this method explicitly; instead it is called by the DUIM-OLE-Server library.

The following example loads three **<integer>** values back from the compound document:

```
define method load-frame-from-storage
  (frame :: <foo-frame>,
   stream :: <storage-istream>) => ()
    frame.foo := istream-read-integer(stream);
    frame.bar := istream-read-integer(stream);
    frame.baz := istream-read-integer(stream);
    values()
end method;
```

The function **istream-read-integer**, page 277, reads a value written by **istream-write-integer**, page 275; while **istream-read-int16**, page 277, reads a value written by **istream-write-int16**, page 276; and **IStream/Read** corresponds to **IStream/Write**.

3.7 The DUIM-OLE-CONTROL module

This section contains a reference entry for the items that the DUIM-OLE-Control module exports from the DUIM-OLE-Control library.

DUIM-OLE-Control also re-exports all the items from the DUIM-OLE-Server and OLE-Control-Framework libraries. See “The DUIM-OLE-SERVER module” on page 124 and “The OLE-CONTROL-FRAMEWORK module” on page 301 for those reference entries.

initialize-ole-control *Macro*

Summary Required macro call for OLE control initialization and registration.

Macro call **`initialize-ole-control ( #rest`** ***options*** **`)`**

Arguments See Description.

Description OLE control applications must call this macro at top-level in order to set up some static initializations necessary for the DLL / OCX's initialization and registration. (It is not an executable expression.)

If your control does not make this macro call, control containers will not be able to connect to it.

You cannot use this macro more than once in a DLL library.

The arguments are keyword options. Note that you must specify **`frame-class:`** and either **`typeinfo:`** or **`class-id:`** and **`title:`**.

`typeinfo:` The type information describing the object. Optional.

If not specified, the library creates default type information based on the attributes of the DUIM sheet being used. If the frame contains a DUIM **`<gadget>`**, dispatch properties and methods corresponding to the gadget protocols will be automatically created (for example, a "value" property for a **`<value-gadget>`**).

If specified, the **`typeinfo:`** value should be an instance of the **`<coclass-type-info>`**, page 186.

If the **`class:`** option of **`<coclass-type-info>`** is specified, it must be a subclass of **`<DUIM-OCX>`**.

class-id: The COM Class ID for the control, as described for **<embeddable-frame>** above.

This is required if **typeinfo:** is not specified; otherwise, this option is not used and the class ID is specified by the **uuid:** option of the **<coclass-type-info>** instead.

value-type: If the **typeinfo:** is being created by default, and the frame contains a **<value-gadget>**, this option can be used to specify the data type of the "value" property of the OLE control. It gets its default value from **gadget-value-type**, which may not be specific enough to map to an OLE Automation type. See Chapter 4, "OLE Automation", for a description of the valid types for a dispatch property.

disp-typeinfo-options:

If the **typeinfo:** is being created by default, this option can be used to add information to the automatically generated dispatch type info. The value should be a sequence of keyword and value pairs such as used as **make** options for a **<disp-type-info>**. For example, to add a user-defined property:

```
disp-typeinfo-options:
  vector(name: "my-dispatch",
         properties: vector(
           make(<variable-description>,
                name: "foo",
                getter: my-property)))
```

frame-class: The class to be instantiated for the DUIM frame. Required. This should be a user-defined subclass of **<embeddable-frame>**.

frame-options: An optional sequence of keyword options to be passed to **make** when instantiating the frame. The default value is an empty sequence. Options used when creating an OLE server, such as **class-id:** and **object-title:**, are not needed here, since they are specified as options to either the **initialize-ole-control** macro or when making the **<coclass-type-info>**.

title: A string to be used as the program name shown in the container's Insert Object dialog. If not provided, the default value is taken from the **documentation:** or **name:** of the type info. This option is required if **typeinfo:** is not specified.

short-title: An optional string used as the program name in container menus and the Links dialog. It must not be more than 15 characters long. If not specified, the default value is taken from the title, truncated to 15 characters if necessary.

name: An optional string used as the name of the class in the constructed type info, if a value for **typeinfo:** is not specified.

If this string is not specified, the default value is taken from the **short-title:** option.

prog-id: The OLE "programmatic identifier". This is an optional string which is used internally and will only be visible in the registry editor. If specified, it must start with a letter; it must not contain any spaces or punctuation except a period (.); and it must not be more than 39 characters long. It must be unique amongst the IDs of all other programs.

If not specified, a default value is created automatically using portions of the title and class ID.

misc-status: An optional **<integer>** formed by using **logior** to combine **$OLEMISC-**… constants to specify various attributes of the OLE control. If not specified, the library attempts to choose an appropriate default for the application frame. See the documentation for **register-ole-server** for more information.

Refer also to the Microsoft OLE/COM API documentation for additional constants and further details.

$OLEMISC-INSIDEOUT

Activate in-place, without any menus or tool bar. More than one such control can be active at the same time.

$OLEMISC-ACTIVATEWHENVISIBLE

Requests that the container always make this control active whenever it is visible. Requires **$OLEMISC-INSIDEOUT** also.

$OLEMISC-ACTSLIKEBUTTON

The control behaves as a button.

$OLEMISC-ACTSLIKELABEL

The control acts as a label for the control following it in the form.

$OLEMISC-NOUIACTIVATE

Control without any user interface. It has no menu, no accelerators, does not need to be activated, and never needs the focus.

$OLEMISC-WANTSTOMENUMERGE

The control wants to merge its menu with the container's.

$OLEMISC-INVISIBLEATRUNTIME

The control has no run-time user interface but should be visible at design time.

$OLEMISC-STATIC

A static object, containing only a presentation without any native data.

versioned-prog-id:

An optional string which is a prog ID that includes the version number — as documented for **register-automation-server**.

versioned-title:

Optional title string that includes the program's version number.

max-storage-size *Open generic function*

Summary
Returns the maximum size, as an integer number of bytes, that will be needed for the data written by **save-frame-to-storage**.

Signature
max-storage-size *frame* => *max-size*

frame	An instance of **<embeddable-frame>**.

Values

max-size	An instance of **<integer>**.

Description
Returns the maximum size, as an integer number of bytes, that will be needed for the data written by **save-frame-to-storage** (for example, count 4 bytes for each call to **istream-write-integer**), or return **#f** if it is not possible to make such an estimate.

You may wish to provide a method specialized on your control's frame class. The default method returns **#f**.

This information is used only if the control container chooses to utilize the **IPersistStream** or **IPersistStreamInit** interface.

<ocx-dispatch> *Open concrete class*

Summary
: Adds some functionality for use in DUIM OLE controls to **`<simple-dispatch>`** from the OLE-Automation library.

Superclasses
: **`<simple-dispatch>`**

Description
: Adds some functionality for use in DUIM OLE controls to **`<simple-dispatch>`** from the OLE-Automation library.

Note that subclasses need to be defined by **`define COM-interface`** instead of **`define class`**.

ocx-frame *Sealed generic function*

Summary
: Given an instance of **`<ocx-dispatch>`** or **`<DUIM-OCX>`**, returns the associated DUIM frame.

Signature
: **`ocx-frame`** ***interface*** **`=>`** ***frame***

Arguments
: *interface* An instance of **`<ocx-dispatch>`** or **`<DUIM-OCX>`**.

Values
: *frame* An instance of **`<embeddable-frame>`**.

Description
: Given an instance of **`<ocx-dispatch>`** or **`<DUIM-OCX>`**, returns the associated DUIM frame.

3.7.1 Getting the ambient properties of the control’s container

To access the ambient properties of the container using it, your OLE control should call **`frame-ole-object`** on its own frame, and then use the facilities described in the documentation for the OLE-Control-Framework library, in “Ambient properties” on page 299.

The following example shows how a control can effectively localize itself according to the control container using it.

```
let obj = frame-ole-object(frame);
let locale-code = OLE-util-locale(obj);
let language = PRIMARYLANGID(locale-code);
select (language)
  $LANG-NEUTRAL,
  $LANG-ENGLISH => …
  $LANG-FRENCH  => …
  $LANG-SPANISH => …
…
```

3.7.2 Automatically generated control properties

When the typeinfo is generated automatically, it provides support for whichever of the following properties and methods are applicable.

Name	Dispatch ID	DUIM function	Applicable to
Value	$DISPID-VALUE	gadget-value	<value-gadget>
items		gadget-items	<collection-gadget>
mode		gadget-selection-mode	<collection-gadget>
Caption	$DISPID-CAPTION	gadget-label	<labelled-gadget-mixin>
Text	$DISPID-TEXT	gadget-text	<text-gadget>
Enabled	$DISPID-ENABLED	gadget-enabled?	Any that accept input
TabStop	$DISPID-TABSTOP		type-union(<tab-control>, <collection-gadget>, <text-gadget>)
ForeColor	$DISPID-FORECOLOR	medium-foreground	<sheet-with-medium-mixin>
BackColor	$DISPID-BACKCOLOR	medium-background	<sheet-with-medium-mixin>

Table 3.1 Default typeinfo properties.

Name	Dispatch ID	DUIM function	Applicable to
activate	$DISPID-DOCLICK	activate-gadget	<action-gadget>

Table 3.2 Default typeinfo methods.

3.8 Low-level DUIM-OLE-Control features

The following can be used for low-level customization; new users need not be concerned with them.

<DUIM-OCX>

Open concrete class

Summary

The class that is instantiated to create the COM object that an OLE control implements.

Superclasses

`<ole-server-framework> <simple-component-object> <IUnknown>`

Description

The class that is instantiated to create the COM object that an OLE control implements. The instance of this class is the "controlling unknown" for the various interfaces supported by the control's COM object.

If you want to define a subclass of this class, you must do so with **`define COM-interface`** rather than **`define class`**.

4

OLE Automation

4.1 Introduction

This chapter explains how to write applications that implement OLE Automation servers and controllers, by using the OLE-Automation library.

4.2 About the OLE-Automation library

The OLE-Automation library allows you to create an application that can be used as an OLE Automation server or controller. The OLE-Automation library includes a high-level framework for OLE Automation as well as a low-level FFI interface to the Microsoft OLE Automation API, as exposed through **`OLE-AUTO.H`**, **`OLEAUT32.LIB`**, and **`OLEAUT32.DLL`**.

Because OLE Automation is a COM technology, some of the names that Automation applications use are actually defined in the Dylan COM library instead of the OLE-Automation library. However, they are all re-exported by OLE-Automation so that you do not need to use COM directly.

By convention, functions from the higher-level part of the library are generally shown beginning with a lower case letter, while lower-level function names begin with an upper case letter, the same as in the Microsoft documentation for use from C or C++, except that "-" is used in place of "_" and a "$" or "?" may be added where appropriate. See Chapter 9, "OLE FFI Facilities" for more on OLE / COM FFI library conventions.

OLE Automation applications do *not* need to use the Dylan OLE library, which is concerned only with implementing compound documents.

4.3 Overview of OLE Automation

This section provides an overview of the concepts and terminology of OLE Automation.

4.3.1 Dispinterfaces, properties, and dispatch methods

Servers expose functionality through *dispatch interfaces* or *dispinterfaces*. Because dispinterfaces are COM interfaces that can be specified in a language-independent fashion, and because all communication between a controller and server is undertaken through dispinterfaces, an Automation controller written in Visual BASIC can access a server written in Dylan without any difficulty.

Dispinterfaces are implemented using the COM interface **`IDispatch`**. In Dylan this is represented by the class **`<simple-dispatch>`**, page 177.

In an Automation *server* written using the OLE-Automation library, a dispinterface is represented as a Dylan object supporting methods that can be accessed by the controller.

In an Automation *controller* written using the OLE-Automation library, a dispinterface is represented by a Dylan object which supports methods for accessing the server.

In OLE Automation, a dispinterface can support two kinds of functionality: *properties* and *methods*. Properties are like Dylan class slots: they have a value that can be read or written. Methods are functions that can be called on the dispinterface implementation object, that can accept additional parameters, and that can return a single result value.

Note: We use the term "dispatch method" to try to distinguish the OLE Automation / COM concept of a method from Dylan's concept of a method.

Every dispinterface can support any number of properties and dispatch methods. Controllers can access these properties and dispatch methods using a *dispatch ID* or *DISPID*, which is an integer index.

Note: Do not assume that this integer index is represented with a Dylan `<integer>` because some servers use larger values.

Each property or dispatch method also has a string name. (References to this name are case insensitive.) A controller can query a server to look up the DISPID corresponding to a name. Some standard properties, called *stock properties*, have a special reserved DISPID, and so do not need to be looked up by name.

4.3.2 Automation servers and COM objects

An Automation server can implement more than one dispinterface, but it will normally implement a single COM object, to which a controller can connect and which it can query to obtain the dispinterfaces it contains.

The description of the COM object as a whole is called a *coclass*, which stands for COM class. Each such kind of object is identified by a *Class ID*. A Class ID is a *GUID* ("Globally Unique IDentifier"), a 128-bit number. (Also sometimes called a *UUID*—"Universally Unique IDentifier").

The following is an example of a GUID that could be used as a Class ID:

```
e90f09e0-43db-11d0-8a04-02070119f639
```

For more on GUIDs, see Section 2.2.5 on page 99.

When an Automation controller wants to use a particular Automation service, it requests an instance of a COM object with a particular Class ID. The Microsoft OLE / COM libraries then arrange for the controller to either connect to an already running server for objects of that Class ID, or start a new server running. It finds out which server to run by looking up the Class ID in the Windows Registry, where it will find the pathname of the server application to be run.

4.3.3 Type libraries and type information

The registry information for a Class ID may also include a *type library*, which provides a description of all of the interfaces that the server supports. Controllers (and various utilities) can examine a COM class's type library without running the associated server application.

For example, the Microsoft OLE Viewer utility can use the type library to display a detailed description of the services provided by an Automation server, and the Visual C++ wizards can use a type library to generate the skeleton for a controller application that could use the server.

Each of the dispinterfaces in the type library is described by an item of *type information*, or a *typeinfo*. There is also an item of *coclass type information* for the COM object that the server implements.

4.3.4 Class factories

When a server starts running, it does not typically create an instance of the COM class it was asked for straight away. Instead, it first creates a *class factory*.

A class factory can create multiple objects of a single COM class. Class factories are COM objects that support the interface **`IClassFactory`** or **`IClassFactory2`**.

The OLE-Automation library provides the class **`<class-factory>`**, page 175. When you create an instance of this class, a COM class factory object is created and registered with Windows. Once you have finished with the class factory (such as when the server is about to terminate) you must end this registration with the function **`revoke-registration`**, page 202.

Once the server's class factory is running, the controller connects to the class factory and directs it to make an instance of the desired COM object.

4.4 Example of Automation

Harlequin Dylan includes two example applications to demonstrate Automation and how to implement it in Dylan. The first application implements an Automation controller that sends commands to the second application, an Automation server.

You can find the controller application in the following folder under your Harlequin Dylan installation:

```
Examples\ole\sample-automation-controller
```

The Automation server is in:

```
Examples\ole\sample-automation-server
```

The server is a simple drawing window that paints shapes upon command from the controller. A user can press buttons on the controller to make the server draw a circle or square, change the paint color, or clear its drawing window.

To use these applications, first open them from the Harlequin Dylan IDE's Examples dialog, where you can find them under the "OLE" section.

Next build the applications with **Project > Build**.

Before you can run them, you need to register the Automation server with Windows so that the controller can connect to it. To register the server, at MS-DOS prompt go to

```
Examples\ole\sample-automation-server\build
```

and call

```
ms-dos> sample-automation-server.exe /RegServer
```

(See "Registering OLE/COM software" on page 101 for more details of registration.)

You should now be able to test the example.

1. Start the **`sample-automation-server.exe`** application.

   The server appears.

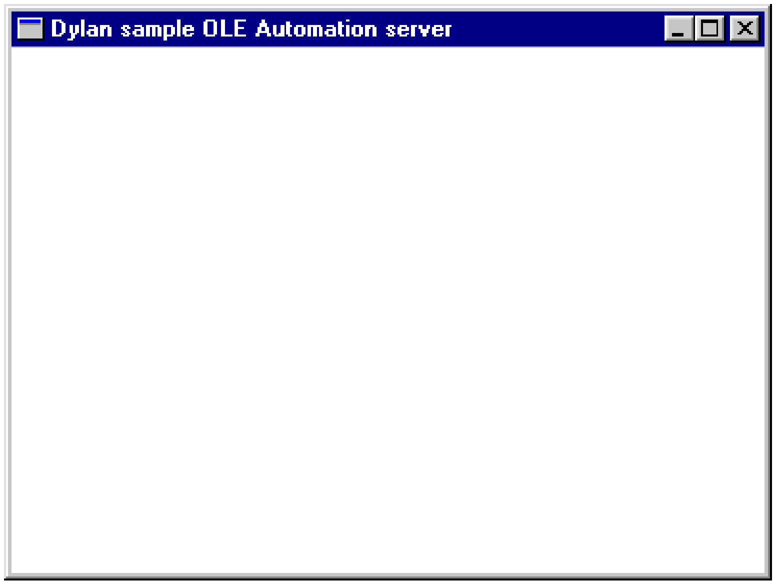

Figure 4.1 Sample Automation server at start-up.

2. Start the **`sample-automation-controller.exe`** application.

   You can find it in the **`build`** subfolder of the folder containing its sources.

   The controller appears.

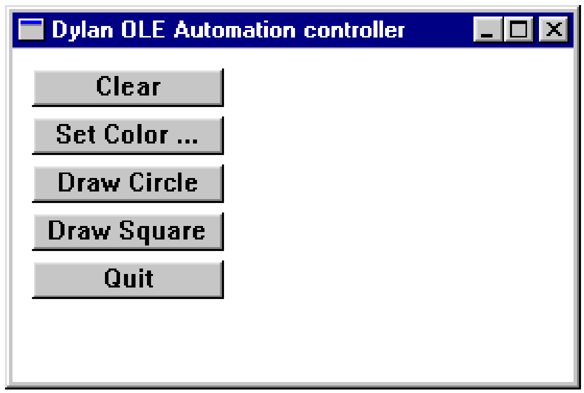

Figure 4.2 Sample Automation controller.

3. Click the **Draw Circle** button in the controller window.

A circle appears in the server window.

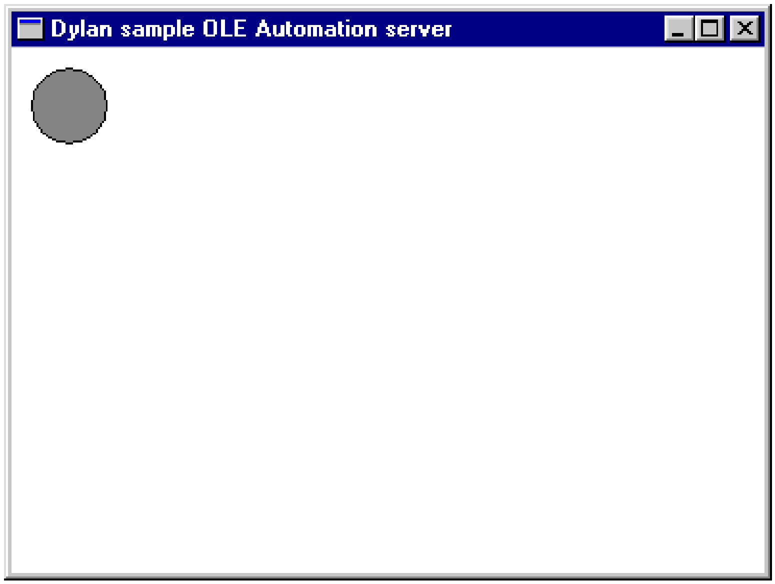

Figure 4.3 Automation server after clicking **Draw Circle** in controller.

The command to draw a circle was sent via the COM connection from the controller to the server, which recognized and responded to it.

4. Try issuing some other commands.

4.5 Building an Automation controller application

The complexity of an Automation controller depends on how many dispinterfaces it uses. This example first assumes a controller that uses the server's default dispinterface, or (equivalently) a server that implements only a single dispinterface. We will discuss the use of multiple dispinterfaces in Section 4.5.9 on page 157.

4.5.1 Initializing OLE Automation

Before using any OLE Automation facilities, a controller application should call the function **`OLE-initialize`**, page 189. This function ensures that the Microsoft OLE/COM libraries have been initialized on the local machine.

The number of calls to this function must be balanced by the same number of calls to the function **`OLE-uninitialize`**, page 190.

4.5.2 Connecting to an Automation server application

Automation controllers connect to Automation servers using the function **`create-dispatch`**, page 200:

```
let disp-interface = create-dispatch(class-ID);
```

This call instantiates a COM object of class *class-ID*, and returns an object representing the default dispinterface implemented by that COM class. The *class-ID* argument is a COM Class ID, as discussed in Section 4.3.2 on page 149.

4.5.3 Getting Dispatch IDs for properties and dispatch methods

Given a string *name*, Automation controllers can use **`get-id-of-name`**, page 193, to return Dispatch IDs for dispinterface properties and dispatch methods:

```
let disp-ID = get-id-of-name(dispinterface, name);
```

4.5.4 Getting and setting the value of a dispatch property

Automation controllers can use **`get-property`**, page 193, to read the value of a property with a known Dispatch ID:

```
let value = get-property(dispinterface, disp-ID);
```

They can use **`set-property`**, page 195, to set the value of a property with a known Dispatch ID to a *new-value*:

```
set-property(dispinterface, disp-ID, new-value);
```

4.5.5 Calling dispatch methods

Automation controllers can call a dispatch method in a dispinterface using **call-simple-method**, page 196:

```
let result =
  call-simple-method(dispinterface, dispatch-method, args, …);
```

The *dispatch-method* can be either a Dispatch ID or a string name. If you expect to call the dispatch method more than once, it is more efficient to use a Dispatch ID here, since the function will otherwise perform the mapping from method name to Dispatch ID each time.

4.5.6 Releasing the dispinterface

When an Automation controller is finished with the dispinterface, it must inform the server. Do this with the function **Release**, page 317:

```
Release(dispinterface);
```

Release is the Dylan implementation of the standard OLE/COM **IUnknown** interface method, **Release**. It is defined on **<IUnknown>** and thus will be inherited by your dispinterface class.

4.5.7 Closing OLE Automation down

Before it exits, an Automation controller application should close OLE down by calling **OLE-uninitialize**, page 190. This function ensures that the Microsoft OLE/COM libraries have been uninitialized on the local machine.

4.5.8 Skeleton OLE Automation controller

The following skeleton code for a simple Automation controller combines the features discussed in sections 4.5.1 to 4.5.7:

```
OLE-initialize();  // Initialize OLE libraries

let dispinterface = create-dispatch(class-id); // Start server

let disp-id =
  get-id-of-name(disp-interface, "frob"); // Look method up
```

```
let result = call-simple-method(dispinterface, disp-id, "foo",
                                "bar"); // Call the method

Release(dispinterface); // Release the server

OLE-uninitialize();  // Shut down OLE
```

A controller application that fits this simple model can be made even simpler using the macro **`with-dispatch-interface`**:

```
with-dispatch-interface dispinterface (class-id)
  let disp-id =
    get-id-of-name(dispinterface, "frob"); // Look method up

  let result = call-simple-method(dispinterface, disp-id, "foo",
                                  "bar"); // Call the method

end with-dispatch-interface;
```

Furthermore, if a DISPID will only be used once, the member name can just be passed directly to **`get-property`** or **`call-simple-method`** instead with no loss of efficiency. So we could do:

```
with-dispatch-interface disp-interface (class-id)
  let result = call-simple-method(disp-interface, "frob", "foo",
                                  "bar");
end with-dispatch-interface;
```

4.5.9 Controllers using more than one dispinterface

If the server provides more than one dispinterface, the first dispinterface that the controller gets back is whatever the server developer designated the default dispinterface. The controller can find the other dispinterfaces by calling the method **`QueryInterface`**, page 317, passing the default dispinterface and the ID (from **`make-GUID`**, page 313) of the desired interface. See also “Servers providing more than one dispinterface” on page 163.

For example:

```
let (status, foo-interface) =
  QueryInterface(dispinterface, interface-id);
```

```
if (FAILED?(status))
  error(…);  // Requested interface not found
else
  … // Use foo-interface
  Release(foo-interface); // Mandatory when finished
end;
```

4.6 Building an Automation server application

This section explains how to build an OLE Automation server application, introducing necessary concepts along the way.

4.6.1 Overview

To create an Automation server in Dylan, you need to provide four things:

1. A Dylan class, and Dylan methods on that class, which implement the dispinterface functionality that you wish to expose to controller applications. Your class must be a subclass of **`<simple-dispatch>`**, page 177.
2. A description of the functionality that the server is making available. This is the *type information* required for the *type library* that potential Automation clients can query and that servers use as part of the implementation of dispatching. See **`<disp-type-info>`**, page 178 and **`<coclass-type-info>`**, page 186.
3. A *class factory* that will instantiate the COM class that the server implements when a controller applications requests it. See **`<class-factory>`**, page 175.
4. Code to register the server so that controller applications can invoke it. See **`register-automation-server`**, page 206.

4.6.2 Defining the Dylan class representing a dispinterface

Note: For now, we will only consider the case of a server that implements a single dispinterface.

You must define the Dylan class representing the dispinterface as a subclass of **`<simple-dispatch>`**, using the macro **`define COM-interface`**. For example:

```
define COM-interface <my-dispatch-object> (<simple-dispatch>)
  slot my-property;
  …
end;
```

The syntax and semantics of the **define COM-interface** macro are exactly the same as **define class**, but for implementation reasons we currently require that you use **define COM-interface** instead. If this should change in a future release, we will of course retain the **define COM-interface** macro for backwards compatibility.

You can define any slots that you like in your dispinterface class, without being concerned about what a controller will be able to access. A controller can access the dispinterface class slot *only* if you explicitly enable it to do so in the type information you provide for the dispinterface.

4.6.3 Defining type information for the dispinterface

To define type information for dispinterface, create an instance of the class **<disp-type-info>**, page 178. We continue the example above by defining type information suitable for **<my-dispatch-object>**.

If you wanted the **my-property** slot in the example above to be an OLE Automation property that a controller can read or write, you could define type information like this:

```
define constant $my-type-info =
  make(<disp-type-info>,
       properties: vector(make(<variable-description>,
                               name: "value",
                               documentation: "some value",
                               getter: my-property,
                               setter: my-property-setter)));
```

The **name:** option specifies the property name that the controller will see; the Dylan class and function names will not be visible externally.

If you wish, you can deny either read or write access by suppling **getter:** or **setter:** values of **#f** respectively.

The **documentation:** is optional; it corresponds what is called a “helpstring” in IDL.

Note that a property does not have to correspond to a slot. It simply needs an accessor method specialized on the class. Any number of properties could be specified as elements of the vector. The DISPID values will be assigned automatically; your server does not need to know what they are, because the controller will ask for them by name.

Note that **`setter: #f`** simply means that the controller cannot change the value—not that it is necessarily a constant. You can specify a property that really is just a constant value like this:

```
…
properties: vector(… make(<constant-description>,
                          name: "pi",
                          value: 3.1416),
                        …
```

In this case, the type information contains the complete implementation of the property; no slot or accessor functions are needed.

4.6.4 Defining dispatch methods for the dispinterface

To define dispatch methods for the dispinterface, you simply define Dylan methods on your dispinterface class. For example:

```
define function foo(this :: <my-dispatch-object>)
  => status :: <SCODE>;
  …
end;

define function bar(this :: <my-dispatch-object>,
                    name :: <BSTR>,
                    #key color :: <BSTR>,
                         size :: <integer>)
  => (status :: <SCODE>, value :: <integer>);
  …
end;
```

4.6.4.1 Dispatch method argument requirements

The first argument of a dispatch method must always be specialized on the dispinterface's Dylan implementation class. The remaining arguments correspond to the dispatch method arguments.

See also “Dispatch method argument and return value types” on page 161.

4.6.4.2 Dispatch method return value requirements

Dispatch methods must always return an OLE status code as their first value. Return the constant **$S-OK** when there is no error. You can return a second value from a dispatch method to represent the result of executing the method. More than two result values are not meaningful.

See also "Dispatch method argument and return value types" on page 161.

The following general purpose error codes could be useful as the first return value from a dispatch method:

$E-UNEXPECTED Unexpected failure.

$E-NOTIMPL Not implemented.

$E-OUTOFMEMORY Ran out of memory.

$E-INVALIDARG One or more arguments are invalid.

$E-POINTER Invalid pointer.

$E-HANDLE Invalid handle.

$E-ABORT Operation aborted.

$E-FAIL Unspecified error.

$E-ACCESSDENIED

General access denied error.

You can convert a Windows error code, such as returned by the Windows function **GetLastError**, to a corresponding **<SCODE>** value with the function **HRESULT-FROM-WIN32**, page 204.

If **<abort>** is signalled during execution of the method (or a property getter or setter), a handler in the OLE-Automation library will catch it and return **$E-ABORT** to the client.

4.6.5 Dispatch method argument and return value types

The argument types and return types for dispatch methods are limited to the following:

```
<integer>, <character>, <single-float>, <double-float>,
<boolean>, <PLARGE-INTEGER>, <ole-array>, <LPDISPATCH>, <class-
factory>,
<machine-word>
```

This limitation is necessary to allow the types to be mapped between Dylan and the C representations of the Microsoft OLE/COM libraries.

If necessary, you can use **pass-as** to constrain the representation of result values.

4.6.6 Defining type information for dispatch methods

The type information for the methods in Section 4.6.4 might look like this:

```
make(<disp-type-info>,
     properties: vector(…),
                 methods:
                   vector(make(<function-description>,
                               function: foo,
                               name: "foo"),
                          make(<function-description>,
                               function: bar,
                               name: "bar",
                               argument-names: #["name", "color",
                                                 "size"],
                               argument-types: vector(<BSTR>,
                                                      <BSTR>,
                                                      <C-long>),
                               result-type: <C-unsigned>)))
```

The **argument-types:** and **result-type:** options are the C type designators if they can not properly be inferred from introspection of the designated **function:**. See **<disp-type-info>**, page 178 for other options you can use.

4.6.7 The DEFINE DISPATCH-INTERFACE macro

Rather than specifying the parts of a dispinterface separately, you can use the high-level macro **define dispatch-interface** to specify both an implementation class and its associated type information. For example:

```
define dispatch-interface <my-dispatch-object>
  (<simple-dispatch>)
  property my-property :: <integer>;
…
```

This defines **my-property** as both a class slot and a dispatch property.

4.6.8 Retaining arguments sent to dispatch methods

When an Automation server receives a string or array as a method argument, it is temporarily allocated a value (an instance of **<ole-array>**, **<BSTR>**, or **<ole-vector>**) that is deleted after the method returns, so you will need to make a copy of the data if you want to keep it. You can use the function **copy-automation-value** to do this. (Copying is done automatically when setting a property.)

Similarly, if an argument is an interface (such as **<class-factory>** or **<LPDISPATCH>**) you must call **AddRef** on it if it will be kept for use after the call returns, and later call **Release** when finished with it.

4.6.9 Cleaning up with TERMINATE

Besides the dispatch methods, your class will also probably need to provide a method for the **terminate** function. This function is called when the last client holding a reference to the dispinterface calls **Release** on it. You can supply a method on **terminate** to clean up data structures or to terminate the server application.

Note that **terminate** is not the same as **finalize** in the Harlequin-Extensions library's Finalization module. A **finalize** method may not be called immediately. Think of the **terminate** function as corresponding to a C++ destructor.

For example:

```
define method terminate (this :: <my-dispatch-object>) => ()
  next-method(); // don't forget this!
  // post ourselves a close message to end the program:
  PostMessage(this.my-window-handle,
              $WM-SYSCOMMAND,
              $SC-CLOSE, 0);
  values()
end method terminate ;
```

4.6.10 Servers providing more than one dispinterface

You can create a server that supports more than one dispinterface as follows.

```
define constant my-coclass-type-info =
  make(<coclass-type-info>,
       uuid: make-GUID(...), // Class ID
       interfaces:
         vector(make(<component-interface-description>,
                     class: <my-dispatch-object1>,
                     typeinfo: make(<disp-type-info>, …)),
                make(<component-interface-description>,
                     class: <my-dispatch-object2>,
                     typeinfo: make(<disp-type-info>,
                                    uuid: make-GUID(…), …), …)
…
));

…

let factory = make-object-factory(my-type-info);
```

Each element of the **interfaces:** sequence specifies a dispinterface with type information, and the Dylan class that must be instantiated to implement it.

By default, the first dispinterface listed will be the default interface that will be returned if a client application calls **QueryInterface** on **$IID-IDispatch** (although it can specify an alternate **uuid:** too).

The type information for the remaining dispinterfaces must specify the **uuid:** option, supplying the ID that a client application can use to select them.

Note too that each type information instance must provide a value for the **name:** option, so that it has a distinguished name in the type library. See "Type libraries and type information" on page 150 a description of a type library.

By default, the class factory instantiates the class **<simple-component-object>**, page 188, but you can specify a user-defined subclass thereof with the **class:** initialization keyword for **<coclass-type-info>**, page 186. This is a subclass of **<IUnknown>**, so its **initialize** method could create additional interfaces using it as the controlling unknown if you want your OLE object to support other interfaces besides **IDispatch**. Also, **<coclass-type-info>** accepts all of the same optional keywords as **<disp-type-info>**, page 178, which can be used to provide type library documentation for the object as a whole. This might be a reason to use it even if there is only one interface. For example:

```
make-object-factory(
  make(<coclass-type-info>,
       name: "foo", documentation: "la de dah",
       major-version: 1, minor-version: 2,
…
```

Alternatively, the type information can be specified by using the macro **define coclass**. For example:

```
define coclass my-coclass
  name "foo";
  uuid make-GUID(...); // Class ID
  default interface <interface-1>;
  interface <interface-2>;
end;
```

where each interface class was defined by **define dispatch-interface**.

4.6.11 Local server initialization

Rather than using **make** to directly instantiate the COM object, a local server should create a class factory. The class **<class-factory>** is provided as a convenient way to do this. For example:

```
make(<class-factory>,
     class: <my-dispatch-object>,
     typeinfo: make(<disp-type-info>, …),
     clsid: make-GUID(…))
```

This code creates and registers a class factory so that the designated dispinterface class is instantiated when a client tries to invoke a server for the designated Class ID. You can also provide any initialization keywords accepted by the class being instantiated, and the following class factory options:

server-context:

Defaults to **$CLSCTX-LOCAL-SERVER**.

connection-flags:

Defaults to **$REGCLS-SINGLEUSE**.

Before your server application terminates, it must call the function **revoke-registration**, page 202, on the class factory.

As a convenience, making a **<class-factory>** and calling **revoke-registration** can be combined by using the macro **with-class-factory** like this:

```
with-class-factory (class: <my-dispatch-object>,
                    typeinfo: make(<disp-type-info>, …),
                    clsid: make-GUID(...) )

… // body of application

end with-class-factory;
```

When using a **<coclass-type-info>**, it contains everything the class factory needs, so the factory can be created simply by doing:

```
let factory = make-object-factory(my-coclass-type-info);
```

4.6.12 Registering an Automation server

Finally, for an Automation controller to be able to invoke an Automation server, the server must be registered in the Windows Registry.

The following code, wrapped around the main program of the server, will take care of registration:

```
if (OLE-util-register-only?()) // [un]register and terminate
  register-automation-server(class-id, prog-id, title, …);
  register-type-library(coclass-type-info);

else
  … // everything else here
end if;
```

If this application is executed with the command-line argument **/RegServer** (case insensitive) it simply records itself in the registry and terminates. Conversely, the command-line argument **/UnregServer** removes the registry entries.

Note: The server application's full pathname is recorded in the registry, so if you move the application to another directory, you will need to register it again.

After an Automation server creates its class factory, it should enter a normal Windows event loop as any other Windows program would. Requests from the client are automatically dispatched by the default message dispatch mechanism.

4.6.13 Skeleton OLE Automation server

Putting all the previous elements together, the main program for an Automation server might look something like this:

```
define constant $my-class-id = make-GUID(…);

define method main-program () => ()

  if (OLE-util-register-only?()) // [un]register and terminate
    register-automation-server($my-class-id,
                               "Foo.Bar",
                               "Simple OLE Automation server");

  else
    initialize-my-application(); // create window etc.

    // create and register the factory object:

    with-class-factory (clsid: $my-class-id,
                        class: <my-dispatch-object>,
                        typeinfo: make(<disp-type-info>, …))

      // normal Windows event loop:

      with-stack-structure (pmsg :: <PMSG>)
        while(GetMessage(pmsg, $NULL-HWND, 0, 0))
          TranslateMessage(pmsg);
          DispatchMessage(pmsg);
        end while;
      end with-stack-structure;

    end with-class-factory;

    values()

  end if;

end method main-program;
```

However, if the program uses DUIM to implement its user interface, the event loop will be provided by DUIM, so the main program will look something like:

```
define method main-program () => ()
  if (OLE-util-register-only?()) // [un]register and terminate
    register-automation-server($my-class-id, "Foo.Bar",
      "a simple OLE Automation server example");
  else
    let frame = make(<my-frame>);
    with-class-factory (clsid: $my-class-id,
                        class: <my-dispatch-object>,
                        typeinfo: ...))
      start-frame(frame);
    end with-class-factory;
  end if;
end method main-program;
```

Some application programs can be used either interactively or by Automation control. In such a case, you might want to create a class factory only if the program was actually initiated by an Automation controller. For this purpose, you can do something like this:

```
…

let factory = #f;

if (OLE-util-automation?())
  factory := make(<class-factory>, …);
end if;

… // body of program

revoke-registration(factory); // does nothing if argument is #f

…
```

4.6.14 In-process server initialization

For an in-process server—one that is built as a DLL file instead of an EXE file—server registration and start-up is handled by code generated by the following macro call, which should appear at the end of the last source file in the server project:

```
in-process-automation-server(
  typeinfo: make(<disp-type-info>, …),
  class-id: $my-class-id,
  prog-id: "Foo.Bar",
  class: <my-dispatch-object>,
  title: "in-process Automation server example");
```

The container application will provide the Windows event loop, so there is no need for one in the server. Any windows that the server uses should be created in the server object's **`initialize`** method and destroyed in its **`terminate`** method.

Do not call **`ExitProcess`** or **`PostQuitMessage`** from an in-process server, because that would terminate the container process.

In-process servers still require registration. To register the DLL file, pass it as the argument to the Windows utility program **`REGSVR32`**. The program supports these options before the file name:

`/s`	Suppress the dialog box that reports completion.
`/c`	Console output instead of dialog box.
`/u`	Unregister instead of register.

4.7 Error reporting

Most of the low-level COM and OLE functions return a status code as their first result value. You can test this value with the functions **`FAILED?`** and **`SUCCEEDED?`** to determine whether an error is indicated. You can also compare the value against various named constants to check for specific error conditions.

These constants are defined in source files under the top-level System folder in the Harlequin Dylan distribution:

```
System\ole\com\com-err.dylan
System\ole\ole-automation\auto-err.dylan
```

The Dylan type of OLE status codes is called either **`<SCODE>`** ("status code") or **`<HRESULT>`** ("result handle"). These two names mean the same thing; the reason for there being two names is historical.

Note: Do not assume that status code is an **`<integer>`**.

In order to simplify your application, the high-level OLE Automation functions do not require you to check status codes. Instead, when an error occurs, it is signalled automatically as an instance of class **`<ole-error>`**, a subclass of **`<error>`**.

If you call a low-level function and want to turn the status code into an error, do something like this:

```
let status = foo(interface, whatever);
check-ole-status(status, "foo", interface, whatever);
```

The call to **check-ole-status** signals an **<ole-error>** if the call to **foo** fails.

4.8 Datatypes

Since the mechanisms of OLE Automation were designed for C and BASIC rather than for Dylan, it does not support the passing of arbitrary Dylan objects. However, instances of the following Dylan classes can be used as method arguments, results, and property values:

```
<integer>, <machine-word>, <byte-character>, <boolean>,
<single-float>, <double-float>, <string>
```

An COM interface can itself be passed as an argument. Also, you can pass instances of **<sequence>** or **<array>** as long as each element is of one of these types. (The elements do not necessarily have to be of the same class if the server is prepared to accept heterogeneous data; that will be true automatically if the server is implemented in Dylan.) For a **<sequence>**, the server will receive the data as a vector, regardless of which subclass of **<sequence>** was used in the controller.

You can also pass the special value **$SQL-NULL** as an argument to indicate an unspecified value. This is intended to correspond to the null value of SQL, and not a C **NULL** pointer.

Note that you cannot pass C pointers, because the controller and server do not necessarily have access to the same address space. Essentially, what is being passed across the dispinterface is a copy of the value.

However, for implementing by-reference parameter passing, it is permissible to pass certain types of C pointers as argument values, but the server can only use the pointer to load or store a value during the duration of the call. Alternatively, a controller can use the convenience functions **out-ref** and **inout-ref** to implement by-reference parameters without direct handling of C pointers.

In some cases, a server in another language may require an argument to be passed with a particular representation even though it cannot be unambiguously inferred from the actual value. A Dylan client can use the function **pass-as** to specify the representation. For example, using the expression **pass-as(<C-long>, 1)** as a method argument forces the value **1** to be passed as a **long** value even though it could fit in a **short**.

4.9 Memory management issues

Some of the values returned by functions in the OLE Automation API are actually allocated by the Win32 libraries, and thus use memory that will not be reclaimed automatically by the Dylan garbage collector.

If a property value or dispatch method result is a string, it will be represented as an instance of class **<BSTR>** ("BASIC string"), which is a subclass of **<string>** (but not a **<byte-string>**). This string value will be allocated by Win32. Similarly, sequences or arrays received as dispatch method results or property values will be instances of **<ole-array>** or **<ole-vector>**, again allocated by Win32.

To ensure that the memory occupied by such a value is recycled, you must call the function **destroy** on the string when it is no longer in use.

Also, if an interface (such as an instance of **<class-factory>** or **<LPDISPATCH>**) is received as a dispatch method result or property value, you must call **Release** on it when finished using it.

4.10 Internationalization

Many of the OLE Automation functions accept an optional **locale:** option that can be used for internationalization. Usually you would not specify it, instead allowing the current user's default locale to be adopted as the default.

If you do need to specify a particular language, construct a locale value with the function **MAKELCID** in the Win32-Common library (see *Win32 API Libraries* in the *C FFI and Win32* reference volume). See also **MAKELANGID** and the associated **$LANG-**… and **$SUBLANG-**… constants. For example, you could do:

```
define constant $british-english-locale =
  MAKELCID(MAKELANGID($LANG-ENGLISH,$SUBLANG-ENGLISH-UK),
           $SORT-DEFAULT);
```

4.11 Low-level OLE Automation API

Besides the high-level utilities described in Section 4.12, the OLE-Automation module also provides a C-FFI interface to the Microsoft OLE Automation API, as exposed through **OLEAUTO.H**, **OLEAUT32.LIB**, and **OLEAUT32.DLL**.

Note that the following names from the Microsoft OLE Automation API names are not supported, because they are considered obsolete:

```
INTERFACEDATA, METHODDATA, PARAMDATA, CreateDispTypeInfo
```

See Chapter 9, “OLE FFI Facilities”, for details of the correspondence between the C/C++ names in the Microsoft OLE documentation and Dylan names in the Harlequin Dylan OLE/COM FFI libraries.

4.12 The OLE-AUTOMATION module

This section contains a reference entry for each item exported from the OLE-Automation module.

<LPDISPATCH> *Open abstract instantiable primary class*

Summary The class of objects that represent OLE Automation **IDispatch** interfaces.

Superclasses **<LPUNKNOWN>**

Description The class of objects that represent OLE Automation **IDispatch** interfaces.

<BSTR> *Sealed concrete primary class*

Summary The class of BASIC strings.

Superclasses **<string>** **<C-Unicode-string>**

Description The class of BASIC strings. Corresponds to the string representation used by OLE Automation property values and dispatch method arguments. The elements are 16-bit Unicode characters.

Besides supporting the full **<string>** protocol, this class can also be regarded as a C pointer, since it is a subclass of the C-FFI type **<C-unicode-string>**.

You can use the constant **$NULL-BSTR** to pass a NULL pointer in place of a string, and the function **null-pointer?** to test a received **<BSTR>** to see if it is a NULL pointer. However, a NULL pointer is treated the same as an empty string by the iteration protocol, so it may not be necessary to do this test.

Note: The Win32 OLE / COM libraries allocate memory on the C heap for these strings. This means that the memory is not automatically reclaimed by the Harlequin Dylan garbage collector. You must call the function **destroy** on a **<BSTR>** when you are finished with it.

You can use the Dylan **as** coercion function to convert between **<BSTR>** and other string types.

The function **copy-as-BSTR** copies any kind of **<string>** to a newly allocated **<BSTR>**; you should use this instead of **as** when returning a value that will be deallocated by the receiving program. The **copy-as-BSTR** function returns **$NULL-BSTR** if the argument is **#f** or a NULL pointer.

<ole-array> *Sealed concrete primary class*

Summary This class allows you to use the Dylan array protocol on data that is actually allocated as a C SAFEARRAY structure.

Superclasses **<array>** **<LPSAFEARRAY>**

Description

This class allows you to use the Dylan array protocol on data that is actually allocated as a C SAFEARRAY structure.

Arrays received by Dylan from an OLE Automation call will appear as an **`<ole-vector>`** if they are one-dimensional, and otherwise as instances of **`<ole-array>`**.

Note: The Win32 OLE/COM libraries allocate memory on the C heap for these arrays. This means that the memory is not automatically reclaimed by the Harlequin Dylan garbage collector. You must call the function **`destroy`** on an **`<ole-array>`** when you are finished with it.

Note: The dimensions in a SAFEARRAY specify both a lower and upper bound for the index, but since Dylan does not support alternate lower bounds, index 0 always corresponds to the first element when viewed as a Dylan array.

<ole-vector>

Sealed concrete primary class

Summary

This class allows you to use the Dylan vector protocol on one-dimensional vectors passed in an OLE Automation call.

Superclasses

`<ole-array>` **`<vector>`**

Description

This class allows you to use the Dylan vector protocol on one-dimensional vectors passed in an OLE Automation call.

You can use the function **`ole-vector`** just like **`vector`** to construct an **`<ole-vector>`** containing the function arguments.

Note: The Microsoft OLE/COM libraries allocate memory on the C heap for these vectors. This means that the memory is not automatically reclaimed by the Harlequin Dylan garbage collector. You must call the function **`destroy`** on an **`<ole-vector>`** when you are finished with it.

<class-factory> *Open concrete primary class*

Summary — This class implements the COM **IClassFactory** interface. Making an instance of it causes it to be registered with the system automatically, for use by potential clients.

Superclasses — **<LPUNKNOWN>**

Init-keywords

clsid: The COM Class ID that identifies the object the server provides. Required. This ID can be represented either as a string of 32 hexadecimal digits in the following format

"{*xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*}"

That is, where each *x* is a hexadecimal digit. For example:

"{e90f09e0-43db-11d0-8a04-02070119f639}"

Alternatively, the ID can be a **<REFCLSID>**, as returned from **make-GUID**.

You can get the ID value by generating a GUID with Harlequin Dylan's **create-id** utility. See "Creating GUID numbers" on page 106.

class: The Dylan class (usually a user-defined subclass of **<simple-dispatch>**) that is to be instantiated when the client (controller) requests it. Required.

args: Optional sequence of initialization arguments to be passed to **make** when instantiating the object. The default is to pass the same arguments as for the **<class-factory>** (**<simple-dispatch>** accepts and ignores those that are only for the factory, and <class-factory> ignores any that it does not recognize.) Note that **<simple-dispatch>** requires **typeinfo:** to be supplied.

`server-module-handle:`

Contains the **`<hModule>`** instance of the server DLL when invoked from an in-process server. This argument is not normally specified by the user, but is passed in automatically when the class factory is created from the in-process DLL entry-point. Note that if you do not specify an explicit **`args`** argument, then this argument will be passed into your object along with all the other initialization arguments.

`server-context:`

Indicates where the server is running. See the description of **`create-dispatch`** below for a list of possible values. Defaults to **`$CLSCTX-LOCAL-SERVER`**. You can use **`$CLSCTX-INPROC-SERVER`** instead to suppress external registration of the factory.

`connection-flags:`

Optional connection flags, controlling whether more than one client is allowed to invoke the same class factory (and hence use the same server process). The value is one of the following OLE constants:

`$REGCLS-SINGLEUSE` – Only one connection allowed. This is the default value.

`$REGCLS-MULTIPLEUSE` – Multiple connections allowed.

`$REGCLS-MULTI-SEPARATE` – Multiple connections allowed, separate control.

For further explanation of these constants, see the Microsoft OLE API documentation for the function **`CoClassRegisterClassObject`**.

Description This class implements the COM **IClassFactory** interface. Making an instance of it causes it to be registered with the system automatically, for use by potential clients.

An Automation server application does not need to use the instance directly, except that it must call the function **revoke-registration** on the instance before terminating.

Any keyword arguments other than those documented above are passed in the **make** call when the Dylan class is instantiated. Note that **<simple-dispatch>** requires you to supply **typeinfo:**.

<simple-dispatch> *Open concrete primary class*

Summary Implements the COM **IDispatch** interface.

Superclasses **<LPDISPATCH>**

Init-keywords **typeinfo:** The COM **ITypeInfo** interface describing the services being provided. This is usually an instance of **<disp-type-info>**. Required.

For supporting multiple locales, the value may be a **<collection>** of **<disp-type-info>** which will be indexed by the locale code.

controlling-unknown:

Designates the **IUnknown** interface if aggregation is being used to provide multiple interfaces from a single OLE object. Optional.

Note that this value is provided automatically if the aggregation is created by using **<coclass-type-info>** and **make-object-**

factory, so it is only in rare cases that you will need to supply a value for this keyword.

Description: Implements the COM **IDispatch** interface.

Automation server applications should define subclasses of this class to represent the dispinterface that they wish to expose to Automation controller applications. You must use **define COM-interface** to define subclasses of this class.

To define dispatch methods, you should define ordinary Dylan methods on your subclass of **<simple-dispatch>** that implement the functionality you wish to expose as dispatch methods.

To define dispinterface properties, you should define type information for your subclass of **<simple-dispatch>**. Do this with **<disp-type-info>**, page 178.

In the OLE/COM model, your dispinterface class is usually instantiated by a class factory. In the OLE-Automation library, the instance of **<class-factory>** (created explicitly or via the **with-class-factory** macro) instantiates your dispinterface subclass, so you do not need to call **make** on it directly.

<disp-type-info> *Sealed concrete class*

Summary: Implements the COM **ITypeInfo** interface.

Superclasses: **<LPUNKNOWN>**

Init-keywords: **name:** A string name for the dispinterface. It should be suitable for use as an identifier in programming languages that might refer to the object. See Description.

uuid: A GUID that identifies the interface. The default is **$IID-IDispatch**.

This should preferably be an instance of **<REFGUID>** (from **make-GUID**) but can also be a string representation of the GUID (containing 32 hexadecimal digits within braces).

properties: A **<sequence>** in which each element is an instance of **<variable-description>** or **<constant-description>** that specifies one dispinterface property. The default is no properties.

methods: A **<sequence>** in which each element is an instance of **<function-description>** that specifies a dispatch method. The default is no dispinterface methods.

documentation: A string describing the dispinterface. This is called a *helpstring* in IDL.

help-file: A Help file name containing help for this dispinterface. The default is no help file.

help-context: The position in the Help file (from **help-file:**) where this dispinterface is documented. Optional.

major-version: Major version number. Default value: 0.

minor-version: Minor version number. Default value: 0.

locale: Internationalization locale. Default value: 0, meaning neutral.

inherit: The base type from which this type information inherits information. Optional.

If specified, the value should be another instance of **<disp-type-info>** which the current type inherits from. The current type will support all of the dispatch properties and dispatch methods of the base type in addition to its own. Some of the other options will gain their default value from the base type where appropriate.

Note: Type information inheritance is implemented separately from the Dylan class hierarchy.

Description

Implements the COM **`ITypeInfo`** interface.

Automation server applications create an instance of this class to represent the type information for each dispinterface class (subclass of **`<simple-dispatch>`**) that they implement.

The string passed to the **`name:`** init-keyword should be suitable for use as an identifier in programming languages that might refer to the object. In particular, if the coclass object is to be used in a Visual Basic form, it will use this name as the class name in the generated Basic source code, and so must be a legal Basic identifier. This means it must begin with a letter; it must not contain any spaces or special characters such as ".", "$", "&", "%", "!", "#", or "@"; it must not be longer than 40 characters; and it must not conflict with a Visual Basic reserved keyword.

<variable-description> *Sealed concrete class*

Summary

Describes a single property of a dispinterface.

Superclasses

`<object>`

Init-keywords

`name:` A string name for the property. Required.

`getter:` A Dylan function that takes the dispinterface as its sole argument and returns the value of the property, or **`#f`** if the property is not externally readable. Default value: **`#f`**.

setter: A Dylan function that changes the value of the property, taking the new value and the dispinterface as its arguments, or **#f** if the client is not permitted to change the property value. Default value: **#f**.

Note that there is a special case. If the value of the property is an interface pointer (such as **<LPUNKNOWN>** or **<LPDISPATCH>**), the setter method must call **AddRef** on the new value and must call **Release** on the old value.

type: A C designator type from the C-FFI library. It is used to specify the datatype that an Automation client written in C or C++ will use to represent the property value. The following values are legal:

<C-int>, **<C-short>**, **<C-long>**, **<C-float>**, **<C-double>**, **<C-signed-char>**, **<C-character>**, **<CY>**, **<DATE>**, **<ole-array>**, **<class-factory>**, **<LPDISPATCH>**, **<C-HRESULT>**, **<VARIANT-BOOL>**, **<VARIANT>**

You can also specify the corresponding Dylan type (such as **<single-float>**, **<double-float>**, **<character>**, or **<boolean>**). The class **<integer>** is considered equivalent to **<C-long>**, and **<string>** is equivalent to **<ole-array>**.

(In addition, the low-level representation as an instance of **<LPTYPEDESC>** can be used here.)

For an array, you must also specify the element type. Do this by using the function **ole-array-type** to construct a type description. For example, **ole-array-type(<C-float>)** for a vector or array of single floats, or **ole-array-type(<VARIANT>)** for a heterogeneous array. Note that dimensions are not specified here.

If you do not specify the **type:**, the OLE-Automation library will select one automatically to most closely match the Dylan type declared for the result value of the getter function. One way or the other, the type must be specified.

Note that the default is the type declared for the generic function. It is not sufficient simply to specify the type for a slot, since there is not enough information here to know which method is applicable.

documentation: A optional string describing the property. This is called a *helpstring* in IDL.

help-context: The position in the WinHelp file (from **help-file:** in the **<coclass-type-info>** instance) where this property is documented. Optional.

disp-id: An **<integer>** which is the DISPID (dispatch ID) by which the client will refer to the property. Usually you would not specify this, and the OLE-Automation library would automatically assign a DISPID. You do need to specify a DISPID when implementing a stock property. For example:

```
make(<variable-description>,
     name: "Value",
     disp-id: $DISPID-VALUE …)
```

Description Describes a single property of a dispinterface.

<constant-description> *Sealed concrete class*

Summary
: Describes a single constant-valued property of a dispinterface.

Superclasses
: **<object>**

Init-keywords

name: A string name for the property. Required.

value: The value of the property. Required.

type: A C type, as for **<variable-description>** above. The default value is derived from the Dylan class of the value.

documentation: An optional string describing the property.

help-context: The position in the WinHelp file (from **help-file:** in the **<coclass-type-info>** instance) where this property is documented. Optional.

disp-id: An **<integer>** which is the DISPID by which the client will refer to the property. Usually you would not specify this, and the OLE-Automation library would automatically assign a DISPID. You do need to specify a DISPID when implementing a stock property. For example:

```
make(<constant-description>,
     name: "Value",
     disp-id: $DISPID-VALUE …)
```

Description
: Describes a single constant-valued property of a dispinterface.

<function-description> *Sealed concrete class*

Summary
: Describes a single dispatch method in a dispinterface.

Superclasses **<object>**

Init-keywords

name: A string name for the dispatch method. Required.

function: The Dylan function or method that implements the functionality of this dispatch method. Required.

This function is called with the dispatch instance as its first argument, and the dispatch method arguments as its remaining arguments. It must return an instance of **<SCODE>** as its first result value, and may optionally return a second result value, which is the value returned from the dispatch method.

The function can have **#key** arguments that correspond to OLE Automation named arguments. An **#all-keys** specification would not be useful.

If the function uses **#rest** but not **#key**, it will be described to a client written in C as accepting a “safe array” as the value of an additional argument.

argument-names:

A **<sequence>** giving the names of the dispatch method arguments. The length of this sequence must match the number of function arguments (including any `#key' arguments) after the dispinterface. Default value: no arguments.

Note that even for **#key** arguments, these names that are exposed to clients do not have to match the Dylan identifiers; the correspondence is strictly by position order. A **#rest** argument does not have a name.

argument-types:

A **<sequence>** giving the C types of the arguments, as for the **type:** option of **<variable-description>** above. The default is to attempt to infer the types from the declared Dylan types of the function arguments.

The sequence can also contain instances of **<ole-arg-spec>**, as returned by **out-ref** or **inout-ref**, to designate a by-reference parameter.

result-type: The C type of the result value, or **#f** if no result. The default value is derived from the declared result values of the Dylan function.

documentation: An optional string describing the function.

help-context: Position in WinHelp file. Optional.

scodes: A **<sequence>** of error code values that might be returned. The default is an empty sequence.

This value is for documentation purposes only: there is nothing to prevent the function from returning other codes not listed here.

flags: **logior** of **$FUNCFLAG-…** values. Default value: 0. (Refer to the Microsoft documentation for the FUNCFLAGS enumeration.) This is not likely to be necessary; it just fine-tunes the appearance of the function in browsers.

disp-id: An **<integer>** which is the DISPID by which the client will refer to the property. If you do not specify one, the OLE-Automation library will automatically assign a DISPID. You should specify a DISPID when the client will expect a particular DISPID, for example for stock properties.

Description Describes a single dispatch method in a dispinterface.

<coclass-type-info> *Sealed concrete class*

Summary Implements the COM **ITypeInfo** interface.

Superclasses **<LPUNKNOWN>**

name: A string name for the COM object class. It should be suitable for use as an identifier in programming languages that might refer to the object. See Description.

uuid: The GUID that identifies the COM object class. Required.

class: The instance of **<class>** that is to be instantiated to implement the COM object. Defaults to **<simple-component-object>**. Usually it would be overridden only by a user-defined subclass of **<simple-component-object>**, but the only requirement is that it implement the **IUnknown** interface and that it accept a **typeinfo:** init argument with the instance of **<coclass-type-info>** as the value.

args: A **<sequence>** of initialization keyword arguments for instantiating the class (in addition to **typeinfo:**). Defaults to the empty sequence. There are no init arguments applicable to **<simple-component-object>**, so use this only if you also specify **class:**.

interfaces: A **<vector>** in which each element is an instance of **<component-interface-description>** describing an interface that is supported by the compound object (COM object).

documentation: A string describing the type library.

This string will appear as the name of the type library in the Microsoft OLE2VIEW utility, which assumes a single line that will be truncated at about 45 characters.

help-file: Name of a Help file containing help for the object class and its interfaces. The default is no help file.

help-context: Position in the Help file where the documentation appears.

major-version: Major version number. Default value: 0.

minor-version: Minor version number. Default value: 0.

locale: Internationalization locale. Default value: neutral.

Description

Implements the COM **ITypeInfo** interface. Automation servers instantiate this class to provide an **ITypeInfo** interface describing a component object class (COM class) that can contain any number of dispinterfaces.

The string passed to the **name:** init-keyword should be suitable for use as an identifier in programming languages that might refer to the object. In particular, if the coclass object is to be used in a Visual Basic form, it will use this name as the class name in the generated Basic source code, and so must be a legal Basic identifier. This means it must begin with a letter; it must not contain any spaces or special characters such as ".", "$", "&", "%", "!", "#", or "@"; it must not be longer than 40 characters; and it must not conflict with a Visual Basic reserved keyword.

<simple-component-object> *Open concrete primary class*

Summary This class implements a compound class or *coclass* object.

Superclasses **`<IUnknown>`**

Description This class implements a compound class or *coclass* object.

You can instantiate it directly, or you can define a subclass of it to instantiate. It takes one required init-keyword, **`typeinfo:`**, whose value is an instance of **`<coclass-type-info>`**, which specifies the component interfaces that will be automatically instantiated when the object is instantiated.

Because this class is a subclass of **`<IUnknown>`**, you can call the function **`QueryInterface`** on an instance of it to obtain one of the component interfaces.

Note that you must define subclasses of this class with the macro **`define COM-interface`** rather than **`define class`**.

<component-interface-description> *Sealed concrete class*

Summary This class is directly instantiated to specify one interface in a "compound class object".

Init-keywords

`typeinfo:`	An **`ITypeInfo`** interface that describes the interface. Usually this will be an instance of **`<disp-type-info>`**. Required.
`class:`	The Dylan **`<class>`** that will be instantiated to implement the interface. The class should be a subclass of **`<simple-dispatch>`**.

args: A **<sequence>** of initialization arguments for the Dylan class, besides the **typeinfo:** and **controlling-unknown:** arguments that are supplied automatically. The default value is the empty sequence.

This is only necessary for keywords added by a user-defined subclass.

flags: Implementation flags. An **<integer>** formed by **logior** of **$IMPLTYPEFLAG-**… values. (Refer to the Microsoft OLE/COM API documentation for the **IMPLTYPEFLAGS** enumeration.) Default value: 0. Meaningful values are:

$IMPLTYPEFLAG-FDEFAULT – Designates this interface as the default. If none have this flag, the first one listed will be used as the default.

$IMPLTYPEFLAG-FSOURCE – A “source” interface, that is, one that is called by the server and should be implemented by the controller.

$IMPLTYPEFLAG-FRESTRICTED – Designates a member that should not be visible or used externally. This is probably not useful in Dylan, since you could simply omit to mention such a member in the type information.

Description This class is directly instantiated to specify one interface in a “compound class object”.

OLE-initialize *Function*

Summary Ensures that the Microsoft OLE/COM libraries have been initialized.

Signature **`OLE-initialize () => ()`**

Arguments None.

Values None.

Description Ensures that the Microsoft OLE/COM libraries have been initialized. Controller applications should call this function before attempting to connect to a server. The function should be used in each thread that will be performing COM/OLE operations (not just once for the whole process). This function returns no values.

Though it does no harm to call this function more than once, you must balance each call to this function by a matching call to the function **`OLE-uninitialize`**.

OLE-uninitialize *Function*

Summary Releases the Microsoft OLE/COM libraries when the controller is finished using them.

Signature **`OLE-uninitialize () => ()`**

Arguments None.

Values None.

Description Releases the Microsoft OLE/COM libraries when the controller is finished using them. You should call this function only to undo a matching call to **`OLE-initialize`**. This function returns no values.

This call allows any resources used by the libraries to be freed, and ensures that any pending messages are handled and any open connections closed.

with-OLE *Macro*

Summary A convenience macro that calls **`OLE-initialize`**, executes some body forms, and calls **`OLE-uninitialize`** in a **`cleanup`** clause.

Macro call

```
with-OLE body end
```

Arguments

body A Dylan body_{bnf}.

Values None.

Description A convenience macro that calls **`OLE-initialize`**, executes the *body* forms, and calls **`OLE-uninitialize`** in a **`cleanup`** clause.

with-dispatch-interface *Statement macro*

Summary Runs a body of code while connected to an Automation server.

Macro call

```
with-dispatch-interface (variable = class-id, #rest keys)
  body
end => values
```

Arguments

variable A Dylan $\text{variable-name}_{\text{bnf}}$.

class-id An instance of **`<string>`** or **`<REFGUID>`**.

keys Instances of **`<object>`**.

body A Dylan body_{bnf}.

Values

values Instances of **`<object>`**.

Description Provides a safe mechanism for working with an Automation server. The macro obtains a dispatch interface pointer to an Automation object, binds it to *variable*, evaluates a *body* of code within the context of this binding, and then releases the interface pointer. The interface pointer is created as if by passing the *class-id* and any *keys* to the function **create-dispatch**.

The values of the last expression in *body* are returned.

copy-automation-value *Function*

Summary Copies a C-allocated value into an equivalent Dylan object.

Signature **copy-automation-value** ***object*** **=>** ***object***

Arguments *object* An instance of **<object>**.

Values *object* An instance of **<object>**.

Description Copies a C-allocated value into an equivalent Dylan object.

If the argument is an instance of **<BSTR>**, **<ole-vector>**, or **<ole-array>**, the value returned is a Dylan object that has the same contents but does not use any C-allocated storage. If the argument is of any other type, it is returned unchanged.

For example, the contents of an **<ole-vector>** would be copied into a **<simple-vector>**, with **copy-automation-value** being recursively called on each element.

A dispatch method could use this method to ensure that an argument value it has received can be kept after the call returns. A controller could use it on a property value or dispatch method result that it receives.

get-id-of-name *Function*

Summary Returns the integer Dispatch ID (DISPID) for the dispatch method or property whose name (for the given *locale*) matches the given string *name*.

Signature **`get-id-of-name`** ***disp-interface name*** **`#key`** ***locale undefined-ok?*** **`=>`** ***disp-ID***

Arguments

disp-interface	An instance of **`<LPDISPATCH>`**.
name	An instance of **`<string>`**.
locale	An instance of **`<integer>`**. Default value: **`$LOCALE-USER-DEFAULT`**.
undefined-ok?	An instance of **`<boolean>`**. Default value: **`#f`**.

Values

disp-ID	An instance of **`<integer>`**.

Description Returns the integer Dispatch ID (DISPID) for the dispatch method or property whose name (for the given *locale*) matches the given string *name*. (The match is not case-sensitive.)

The *disp-interface* argument is an instance of **`<LPDISPATCH>`**.

This function works for dispatch methods or properties, but does not support mapping the names of the arguments of dispatch methods.

If *name* is not defined and *undefined-ok?* is true, the function returns the integer value **`$DISPID-UNKNOWN`** instead of signalling an error.

get-property *Function*

Summary Returns the value of a property of a dispinterface.

Signature **get-property** ***disp-interface property*** **#key** ***default locale index***
=> ***value***

Arguments

disp-interface An instance of **<LPDISPATCH>**.

property An instance of **<string>** or **<disp-id>**.

The class **<disp-id>** is defined as **type-union(<integer>, <machine-word>)**. Usually, a dispatch ID value will be an **<integer>**, but occasionally it will not fit.

default An instance of **<object>**.

locale An instance of **<integer>**. Default value: **$LOCALE-USER-DEFAULT**.

Values

value An instance of **<object>**.

Description

Returns the value of a property of a dispinterface.

The *disp-interface* argument is an instance of **<LPDISPATCH>**.

The *property* argument is either the name of the property (an instance of **<string>**), or the DISPID number. If the property is to be referenced more than once, it is much more efficient to use **get-id-of-name** to map the name to a DISPID just once.

For accessing an indexed property, use the *index* keyword option to specify either a single index or a **<sequence>** of index values. If the designated property is not supported, and the *default* keyword option is specified, the value passed with that keyword is returned. This is useful for querying stock properties. Otherwise, the function signals an error. For some servers, the default value will also be used for an out-of-range index.

You can also use this function on the left-hand side of an assignment to set the value of a property.

get-indexed-property *Function*

Summary **get-indexed-property** ***disp-interface disp-id*** **#rest** ***indexes***
=> ***value***

Arguments

disp-interface An instance of **<LPDISPATCH>**.

disp-id An instance of **<disp-id>**.

The class **<disp-id>** is defined as **type-union(<integer>, <machine-word>)**. Usually, a dispatch ID value will be an **<integer>**, but occasionally it will not fit.

indexes Instances of **<object>**.

Values

value An instance of **<object>**.

Description

This function provides a simpler way to access and indexed property than **get-property** when the locale and default value options are not needed. For example,

```
get-indexed-property(intf, disp-id, x, y)
```

is equivalent to:

```
get-property(intf, disp-id, index: vector(x, y))
```

This function can also be used on the left side of an assignment to set an element of an indexed property. Note that here the call to **get-property** uses a keyword **index:** that is otherwise undocumented. You can use this keyword to specify either a single index or a **<sequence>** of index values.

set-property *Function*

Summary Sets the value of a property in a dispinterface.

Signature **set-property** ***disp-interface property new-value*** **#key** ***locale***
=> ()

Arguments

disp-interface An instance of **<LPDISPATCH>**.

property An instance of **<string>** or **<disp-id>**.

The class class **<disp-id>** is defined as **type-union(<integer>, <machine-word>)**. Usually, a dispatch ID value will be an **<integer>**, but occasionally it will not fit.

new-value An instance of **<object>**.

locale An instance of **<integer>**. Default value: **$LOCALE-USER-DEFAULT**.

Values

None.

Description

Sets a property in *disp-interface* to *new-value*. As for the function **get-property**, the *property* argument is either a string name or integer DISPID, and *disp-interface* is an instance of **<LPDISPATCH>**.

call-simple-method *Function*

Summary

This function provides a simple way to call a dispatch method in a dispinterface.

Signature

call-simple-method ***disp-interface dispatch-method*** **#rest** ***args*** **=>** ***result***

Arguments

disp-interface An instance of **<LPDISPATCH>**.

dispatch-method An instance of **<string>** or **<disp-id>**.

The class class **<disp-id>** is defined as **type-union(<integer>, <machine-word>)**. Usually, a dispatch ID value will be an **<integer>**, but occasionally it will not fit.

	args	Instances of **`<object>`**.
		Each *args* value can be either the actual value to be passed in the dispatch method call, or an instance of **`<ole-arg-spec>`** from the functions **`pass-as`**, **`out-ref`**, or **`inout-ref`**.
Values	*result*	An instance of **`<object>`**.

Description

This function provides a simple way to call a dispatch method in a dispinterface.

This function is limited to dispatch methods that have only positional arguments (as opposed to named arguments) and which do not need to have a locale specified.

The *dispatch-method* argument is either the name of the dispatch method (a **`<string>`**), or its DISPID. If you expect to call the dispatch method more than once, it is much more efficient to use **`get-id-of-name`** to map the name to a DISPID just once.

pass-as *Function*

Summary **`pass-as`** ***type value*** **`=>`** ***representation***

Arguments	*type*	An instance of **`<integer>`** or **`<type>.`**
	value	An object that can be represented as **`<type>`**.
Values	*representation*	An instance of **`<ole-arg-spec>`**.

Description

Use this in a client as an argument to functions such as **`call-simple-method`** or **`set-property`** or in a server as a return value from a property or method to specify passing the given *value* using the representation designated by *type*. The return value is an instance of **`<ole-arg-spec>`**.

The *type* may be specified either as one of the low-level VARIANTARG type codes, such as **$VT-I2** or **$VT-I4** or as a C-FFI type designator, such as **<C-short>** or **<C-long>**.

out-ref *Function*

Summary **out-ref** ***type*** **=>** ***ref***

Arguments *type* An instance of **<integer>** or **<type>**.

Values *ref* An instance of **<ole-arg-spec>**.

Description Use this in a client to construct an object that can be passed as an argument to **call-simple-method** and receive the value of a returned output parameter. The return value is an instance of **<ole-arg-spec>**.

The *type* of the value is specified as either one of the low-level VARIANTARG type codes (such as **$VT-I4**) or as a C-FFI type designator, or as a corresponding Dylan type. Use the accessor **arg-spec-value** to get the value after the call.

For example, if a server defines a method that has a by-reference output parameter with C type **long**, a Dylan client could receive the value like this:

```
// first make a place to hold the output parameter
let ref = out-ref(<C-long>);
// then call the server method
call-simple-method(disp-interface, disp-id, ref);
// now pick up the received value
let value = arg-spec-value(ref);
```

If the server is written in Dylan, it will simply receive an instance of **<C-long*>**, and should use **pointer-value-setter** to store the value.

inout-ref

Function

Summary **inout-ref** ***value*** **#key** ***vt*** ***type*** **=>** ***ref***

Arguments

value An instance of **<object>**.

vt Optional. An instance of **<integer>**.

type Optional. An instance of **<type>**.

Values

ref An instance of **<ole-arg-spec>**.

Description

This is similar to **out-ref** above, except that it is for input-output parameters. The *value* argument is the value to be passed in (may be changed by **arg-spec-value-setter** if the reference is to be used for more than one call). The type is specified by either the *vt* option with a VARIANTARG type code or the *type* option with a C or Dylan type designator. If neither *vt* or *type* is given, the type will be inferred from the *value*.

The return value is an instance of **<ole-arg-spec>**.

arg-spec-value

Function

Summary **arg-spec-value** ***ref*** **=>** ***value***

Arguments

ref An instance of **<ole-arg-spec>**.

Values

value An instance of **<object>**.

Description

Accessor to return the value from an **<ole-arg-spec>** reference object created by the **out-ref**, **inout-ref**, or **pass-as** functions. May also be used on the left-hand side of an assignment to change the value.

create-dispatch *Function*

Summary Invokes or connects to an Automation server, as specified by its COM Class ID.

Signature **create-dispatch** ***class-ID*** **#key** ***context interface-ID*** **=>** ***disp-interface***

Arguments

class-ID	An instance of **<string>** or **<REFGUID>**.
context	An instance of **<integer>**. Default value: **$CLSCTX-ALL**.
interface-ID	An instance of **<REFCLSID>** or **<string>**. See Description.

Values

disp-interface	An instance of **<LPDISPATCH>**.

Description Invokes or connects to an Automation server, as specified by its COM *class-ID*. This is a function for use by Automation controllers.

Normally, this function returns the server's primary dispinterface, but you can use the *interface-ID* option to select a specific interface.

You should call **Release** on the returned interface when finished using it.

The *class-id* argument and *interface-ID* option may be either an instance of **<REFCLSID>** (such as returned by **make-GUID**) or a string of 32 hexadecimal digits in the following format

"{*xxxxxxxx*-*xxxx*-*xxxx*-*xxxx*-*xxxxxxxxxxxx*}"

That is, where each *x* is a hexadecimal digit. For example:

```
"{e90f09e0-43db-11d0-8a04-02070119f639}"
```

The *context* option takes one of the following values, constraining which implementation of the server to use:

$CLSCTX-INPROC-SERVER

An in-process server (DLL running in the container's process).

$CLSCTX-LOCAL-SERVER

A server running in its own process (EXE file).

$CLSCTX-REMOTE-SERVER

Server running on a remote machine.

$CLSCTX-SERVER

Any of the above.

$CLSCTX-INPROC-HANDLER

An in-process handler.

$CLSCTX-ALL

Any of the above. This is the default value.

If the Class ID is not found in the Windows registry, or there is no server that matches the requested *context*, the function will signal an **<ole-error>**, with status code of **$REGDB-E-CLASSNOTREG**.

If an interface matching the *interface-ID* is not found, the function will signal an **<ole-error>** with status code **$E-NOINTERFACE**.

make-object-factory *Function*

Summary: Creates, registers, and returns a COM class factory for a COM object.

Signature: **make-object-factory** ***typeinfo*** **#rest** ***other-args*** **=>** ***factory***

Arguments: *typeinfo* — An instance of **<coclass-type-info>**.

	args	Initialization arguments to pass to **`make`** on **`<class-factory>`**.
Values	*factory*	An instance of **`<class-factory>.`**

Description

Creates, registers, and returns a COM class factory for the COM object described by the given coclass type information, *typeinfo*. Automation servers should use this function to allow repeated instantiations of the COM objects they support.

The additional arguments are used as initialization keywords for either the instance of **`<class-factory>`** or the COM object implementation class.

You must call **`revoke-registration`** on the returned *factory* object before your program terminates.

See Section 4.3.4 on page 150 for more information about the role of class factories in OLE Automation.

revoke-registration *Function*

Summary

Unregisters a COM class factory.

Signature

`revoke-registration` ***factory*** **`=> ()`**

Arguments	*factory*	An instance of **`<class-factory>`**.

Values

None.

Description

Unregisters a COM class factory, making it no longer available to potential controller clients. This function returns no values.

This function does nothing if the argument is **`#f`** or **`$null-interface`**, or if it has already been called on the same factory instance.

See Section 4.3.4 on page 150 for a description of the role of class factories in OLE Automation.

with-class-factory *Macro*

Summary Instantiates a **`<class-factory>`**, executes the body forms, and finally calls **`revoke-registration`** on the class factory instance in a cleanup clause.

Macro call

```
with-class-factory (args)
  body
end
```

Arguments

args	Initialization arguments to pass to **`make`** on **`<class-factory>`**.
body	A Dylan body$_{\mathrm{bnf}}$.

Description This convenience macro calls **`make`** on **`<class-factory>`**, with the given initialization keyword arguments, then executes the *body* forms, and finally calls **`revoke-registration`** on the class factory instance in a cleanup clause.

See Section 4.3.4 on page 150 for a description of the role of class factories in OLE Automation.

ole-array-type *Function*

Summary **`ole-array-type`** ***element-type*** **`=>`** ***array-type-description***

Arguments *element-type*

Values *array-type-description*

Description Creates and returns a pseudo-type used to specify array types for the **`type:`**, **`argument-types:`** or **`result-type:`** options when creating a type information instance (an instance of **`<disp-type-info>`**).

The argument is the type descriptor (C type or Dylan class) of the elements of the array. If the argument is **`<VARIANT>`** or **`<object>`**, a heterogeneous array is indicated, in which each element carries its own type information at run time.

HRESULT-FROM-WIN32 *Function*

Summary **`HRESULT-FROM-WIN32`** ***error-code*** **`=>`** ***status***

Arguments *error-code* An instance of **`<integer>`**.

Values *status* An instance of **`<SCODE>`**.

Description Given an integer Windows error code, such as returned by Windows function **`GetLastError`**, return the corresponding **`<SCODE>`** value.

OLE-util-in-process-startup? *Function*

Signature **`OLE-util-in-process-startup?() =>`** ***in-process?***

Arguments None.

Values *in-process?* An instance of **`<boolean>`**.

Description Can be used when a server can be built as either an in-process (DLL) or a local (EXE) server. Rather than building the configuration information into the sources, you can use this function to determine at run-time which configuration is being used. Typically, you would avoid invoking your application's main routine in the in-process case:

```
unless (OLE-util-in-process-startup?())
  main-program()
end;
```

OLE-util-automation? *Function*

Summary `OLE-util-automation? () =>` ***automation?***

Arguments None.

Values *automation?* An instance of `<boolean>`.

Description **Note:** Not applicable to in-process servers.

Returns `#t` if the application was invoked with the command-line option `/Automation` (case-insensitive), indicating that execution was initiated by an Automation controller client, as opposed to being invoked directly by a user.

Your application might want to make a class factory *only* if this is true. For example:

```
…
let factory = #f;
if ( OLE-util-automation?() )
  factory := make(<class-factory>, …);
end if;

… // body of program

revoke-registration(factory); // no-op if arg is #f
…
```

OLE-util-file-arg *Function*

Summary `OLE-util-file-arg () =>` ***file-name*** `::`

Arguments None.

Values *file-name* An instance of **`false-or(<string>)`**.

Description This convenience function can be used by applications that want to accept a single file name as a command line argument. If such an argument has been supplied, it will be returned as a string.

OLE-util-register-only?

Function

Summary **`OLE-util-register-only? () =>`** ***register?***

Arguments None.

Values *register?* An instance of **`<boolean>`**.

Description **Note:** Not applicable to in-process servers.

Returns **`#f`** unless the application's first command-line argument is any of **`/RegServer`**, **`-RegServer`**, **`/UnregServer`** or **`-UnregServer`**. The comparison of the command-line arguments with these strings is not case-sensitive.

If the result is not false, the server application should just call **`register-automation-server`** and exit without doing anything else.

register-automation-server *Function*

Summary **`register-automation-server`** ***class-ID prog-ID title-string*** **`#key`** ***versioned-prog-ID versioned-title*** **`=> ()`**

Arguments *class-ID* An instance of **`<string>`** or **`<REFGUID>`**.

prog-ID An instance of **`<string>`**.

title-string An instance of **<string>**.

versioned-prog-ID

An instance of **false-or(<string>)**.

versioned-title An instance of **false-or(<string>)**.

Values None.

Description If the application was invoked with the command-line argument **/RegServer** or **-RegServer**, this function creates the Windows Registry entries necessary for the current application to be invoked by an Automation controller. If the application was invoked with **/UnregServer** or **-UnregServer**, this function attempts to delete any registry entries belonging to the application.

This function returns no values.

The *class-ID* argument must be the same value as passed as the **clsid:** argument to **make** on **<class-factory>**, or as the **uuid:** argument to **make** on **<coclass-type-info>**.

Note that if you want to be able to have two versions of a program installed at the same time, then both versions must have different Class IDs, and you must also specify a **versioned-prog-id:** for each.

The *prog-id* argument must be the class's programmatic identifier or Prog ID. A Prog ID is a unique symbolic alias for a numeric GUID, represented by a string. (Prog IDs allow a server to be used by Visual Basic and are also required for compatibility with OLE1.) A Prog ID must begin with a letter; it cannot contain any spaces or punctuation except the period (.) character, and it must not be more than 39 characters long. In order to ensure uniqueness, the recommended format is:

<vendor-name>.<product-name>

For example:

```
AcmeWidgets.FrobMaster
```

The *title-string* identifies the program in displays to its user. It should not include the version number.

The optional *versioned-prog-id* must be either **#f** or a string. We recommend that it be the same as the *prog-ID*, with a version identification appended. For example:

```
AcmeWidgets.FrobMaster.2.1
```

The optional *versioned-title* must be either **#f**, or a title string that includes the program's version number.

register-type-library *Function*

Summary **register-type-library *typeinfo* => ()**

Arguments *typeinfo* An instance of **<ITypeInfo>**.

Values None.

Description Creates an OLE/COM *type library* file from the given type information and records it in Windows Registry, associating it with this Automation server.

If the server was invoked with the command line option **/UnregServer** or **-UnregServer**, this function attempts to delete the type library from the registry.

This function returns no values.

The type library file that is created is placed in the same directory as the server (.EXE) file, and will have the same name but with the version and locale numbers appended, and the extension .TLB. For example, **foo.exe** might become **foo-2-1-0.tlb**.

Automation controllers and other utilities can use the type library to find out what facilities are provided by the Automation server the type library describes, without actually executing the server.

It is not mandatory to create and register a type library for your Automation server, since the server itself does not use it, and an Automation controller does not necessarily need it either. If a connected controller requests the type library, it will use the registered one if it exists, or else force the type library to be created and registered dynamically.

Normally the argument should be an instance of **`<coclass-type-info>`**, although if there is just a single dispinterface, it is also possible to pass an instance of **`<disp-type-info>`** provided it was created with a unique **`uuid:`** argument (not just **`$IID-IDispatch`**); it will then act as both an Interface ID and a Class ID.

The type library can be identified by its version number and locale as well as its Class ID (type info options **`major-version:`**, **`minor-version:`**, **`locale:`**, and **`uuid:`**, respectively) so different versions of an application can be installed at the same time, and one application can register multiple type libraries with different locales (but the same Class ID, version, and help directory). (However, there will still be only one call to **`make-object-factory`**, since it does not depend on the locale.)

in-process-automation-server *Macro*

Summary In-process Automation servers must call this macro at top-level in order to set up some static initializations needed for DLL initialization and registration.

Macro call **`in-process-automation-server #key`** ***typeinfo title class-ID prog-ID class args versioned-prog-ID versioned-title***

Arguments

typeinfo An instance of **`<disp-type-info>`** or **`<coclass-type-info>`**. Required.

title An instance of **`<string>`**. Default value: see Description.

class-ID An instance of **`<REFGUID>`** or **`<string>`**. Default value: see Description.

prog-ID An instance of **`<string>`**. Default value: see Description.

class The Dylan implementation class to be instantiated. Default value: see Description.

args An instance of **`<sequence>`**. Default value: see Description.

versioned-prog-ID

An instance of **`<string>`**. Default value: see Description.

versioned-title An instance of **`<string>`**. Default value: see Description.

Description

Note: Not applicable to local servers.

In-process Automation servers must call this macro at top-level in order to set up some static initializations needed for DLL initialization and registration. (It is not an executable expression.)

The macro expansion provides definitions for the Windows functions **`DllRegisterServer`**, **`DllUnregisterServer`**, **`DllGetClassObject`**, and **`DllCanUnloadNow`**. Without this, the container application will be unable to connect to the server.

This macro cannot be used more than once in a DLL library. The arguments are:

typeinfo: The ITypeInfo interface that describes the services being provided. This should be an instance of **<disp-type-info>** or **<coclass-type-info>**. Required.

title: String to appear in a container application's Insert Object dialog box, to identify this server application.

The value passed to the **typeinfo:** parameter contains a documentation string, probably provided by a keyword when creating the typeinfo. The value of **title:** defaults to that string.

prog-id: The "programmatic identifier" string, as described for **register-automation-server**. Required.

class-id: The COM class ID of the server object. Optional; defaults from the typeinfo.

class: The class that should be instantiated when the server is invoked. Optional; defaults from the typeinfo if it is a coclass; otherwise, the class defaults to **<simple-dispatch>**.

args: Optional **<sequence>** of arguments to be passed to **make** when instantiating the class. Defaults from the typeinfo if it is a coclass, otherwise, defaults to an empty sequence.

versioned-prog-id:

Optional string that is recommended to be the same as the Prog ID with a version identification appended.

versioned-title:

Optional title string that includes the program's version number.

pass-as *Function*

Summary **pass-as** ***type value*** **=>** ***representation***

Arguments

type An instance of **<integer>** or **<type>.**

value An object that can be represented as **<type>**.

Values

representation An instance of **<ole-arg-spec>**.

Description

Use this in a client as an argument to functions such as **call-simple-method** or **set-property** or in a server as a return value from a property or method to specify passing the given *value* using the representation designated by *type*. The return value is an instance of **<ole-arg-spec>**.

The *type* may be specified either as one of the low-level VARIANTARG type codes, such as **$VT-I2** or **$VT-I4** or as a C-FFI type designator, such as **<C-short>** or **<C-long>**.

out-ref *Function*

Summary **out-ref** ***type*** **=>** ***ref***

Arguments

type An instance of **<integer>** or **<type>**.

Values

ref An instance of **<ole-arg-spec>**.

Description

Use this in a client to construct an object that can be passed as an argument to **call-simple-method** and receive the value of a returned output parameter. The return value is an instance of **<ole-arg-spec>**.

The *type* of the value is specified as either one of the low-level VARIANTARG type codes (such as **$VT-I4**) or as a C-FFI type designator, or as a corresponding Dylan type. Use the accessor **arg-spec-value** to get the value after the call.

For example, if a server defines a method that has a by-reference output parameter with C type **long**, a Dylan client could receive the value like this:

```
// first make a place to hold the output parameter
let ref = out-ref(<C-long>);
// then call the server method
call-simple-method(disp-interface, disp-id, ref);
// now pick up the received value
let value = arg-spec-value(ref);
```

If the server is written in Dylan, it will simply receive an instance of **<C-long*>**, and should use **pointer-value-setter** to store the value.

inout-ref *Function*

Summary **inout-ref** ***value*** **#key** ***vt*** ***type*** **=>** ***ref***

Arguments

value	An instance of **<object>**.
vt	Optional. An instance of **<integer>**.
type	Optional. An instance of **<type>**.

Values

ref	An instance of **<ole-arg-spec>**.

Description

This is similar to **out-ref** above, except that it is for input-output parameters. The *value* argument is the value to be passed in (may be changed by **arg-spec-value-setter** if the reference is to be used for more than one call). The type is specified by either the *vt* option with a VARIANTARG type code or the *type* option with a C or Dylan type designator. If neither *vt* or *type* is given, the type will be inferred from the *value*.

The return value is an instance of **`<ole-arg-spec>`**.

arg-spec-value *Function*

Summary **`arg-spec-value`** ***ref*** **`=>`** ***value***

Arguments *ref* An instance of **`<ole-arg-spec>`**.

Values *value* An instance of **`<object>`**.

Description Accessor to return the value from an **`<ole-arg-spec>`** reference object created by the **`out-ref`**, **`inout-ref`**, or **`pass-as`** functions. May also be used on the left-hand side of an assignment to change the value.

5

Macros for Defining Custom Interfaces

5.1 Introduction

This chapter provides reference documentation for OLE-Automation and COM library macros that can be used to define custom interfaces with Harlequin Dylan.

5.2 Overview

The following COM and OLE-Automation macros define custom interfaces:

- **`define custom-interface`** (custom interface definer)
- **`define coclass`** (component object definer)
- **`define vtable-interface`** (v-table interface with type info)
- **`define dispatch-client`** (client of dispatch interface)
- **`define dispatch-interface`** (dispatch interface server and type info)
- **`define dual-interface`** (combined v-table and dispatch interface)

5.3 Macros for defining custom interfaces

This section contains reference material for the COM and OLE-Automation macros used to define custom interfaces with Harlequin Dylan.

define custom-interface *Definition macro*

Summary Creates a new custom v-table COM interface.

Macro call

```
define [ class-adjectives ] custom-interface interface-name
 (superclass)
  [ client-class client-class-name
  [ :: client-superclass-name ] ; ]
    uuid uuid;
  { property | function }*
end [ custom-interface ] [ interface-name ]
```

Arguments

class-adjectives	Any Dylan class definition adjective.
interface-name	$name_{bnf}$
superclass	$name_{bnf}$
client-class-name	$name_{bnf}$
uuid	A GUID that identifies the interface.

The default is **$IID-IDispatch**.

This value should preferably be an instance of **<REFGUID>** (from **make-GUID**) but can also be a string representation of the GUID (containing 32 hexadecimal digits within braces).

property

```
{ constant property property-name :: property-type ;
| constant property property-name [ :: property-type ]
    = property-value ;
| [ virtual ] property property-name :: property-type ;
}
```

function

```
{ function | vtable-member } member-name
    ( { argument-name :: argument-type, }* )
      => ( { result-name :: result-type, }* );
```

Description

Used to create a new COM interface, with a v-table determined by the superclass and the **function**, **vtable-member** and **property** clauses. An implementation of the interface can be created by using **define COM-interface** to define a subclass of *interface-name*.

The *superclass* specifies the interface class to inherit from. Often it will be **<IUnknown>**, but it could be another standard interface (for example, **<IDispatch>** for a dual interface) or another class defined by **define custom-interface**.

The optional **client-class** clause specifies the name that will be defined as the C pointer class to be used by a client. If a *client-superclass-name* is supplied, it is used as the superclass for the client class; otherwise **<C-interface>** is used as the superclass. After obtaining an interface pointer from a function such as **QueryInterface** or **CoCreateInstance**, you should call **pointer-cast** to convert it to this class since that is where the client-side methods will be defined. If a client-class clause is not specified, client methods will be defined. If a client-class clause is not specified, client methods will be defined on **<C-interface>**. Omitting client-class should be done with caution, because it can lead to name conflicts between members of different interfaces, as well as losing some type safety. (Invoking a v-table member through a wrong pointer can cause an out-of-language crash.)

The **uuid** clause is required; it specifies the interface ID by which **QueryInterface** will find this interface. The value may be given as either a string or a structure created by **make-GUID**.

A **`function`** or **`vtable-member`** declaration adds a slot to the v-table, defines a method on the **`<C-interface>`** subclass that will do an indirect call through the v-table slot, and a **`C-callable-wrapper`** that will connect the v-table slot of a Dylan instance to a call to the generic function on the Dylan object. The actual implementation method must be defined separately. The method must accept an instance of *interface-name* and any additional arguments specified in the declaration. For consistency with **`define dispatch-interface`**, the **`function`** clause implies that the first return value is an **`<SCODE>`** status code, followed by the explicit return values. The **`vtable-member`** clause differs only in that all result values must be declared explicitly; thus it allows specifying methods that do not return a status code.

The first return value will be implemented directly as the return value of the C function, while any additional return values will be converted to output parameters following the specified input parameters.

A **`property`** declaration is an abbreviation for defining a pair of methods to get and set a value. For example,

```
property blip :: <integer>;
```

is equivalent to the pair

```
vtable-member blip () => (value :: <integer>);
vtable-member set_blip ( value :: <integer> ) => ();
```

plus defining a client-side method on **`blip-setter`** that calls **`set_blip`** and a server-side method on **`set_blip`** that calls **`blip-setter`**, so that the internal name **`set_blip`** never needs to be referenced directly. (This conversion is necessary because ...**`-setter`** methods require the opposite argument order.)

Thus, an implementation for the property can be provided by simply defining a slot with the same name in the implementation class. The optional modifier **`virtual`** has no effect here; it is allowed simply for consistency with **`define dispatch-interface`**.

For a **`constant property`**, only the getter member is created. If a *property-value* is specified, a getter method that returns that value will be created automatically on the server side.

The types of properties, function arguments, and function results may be specified by using either one of the following Dylan classes:

`<integer>, <single-float>, <double-float>, <string>, <byte-string>, <machine-word>, <SCODE>, <HRESULT>, <character>, <byte-character>, <byte>, <boolean>, <Interface>, <mapped-interface>, <LONG>, <LPUNKNOWN>, <BSTR>`

or one of the following C types:

`<C-boolean>, <C-both-signed-long>, <C-both-unsigned-long>, <C-byte>, <C-char>, <C-character>, <C-double>, <C-float>, <C-HRESULT>, <C-int>, <C-long>, <C-raw-unsigned-long>, <C-short>, <C-signed-char>, <C-string>, <C-unicode-string>, <C-unsigned-char>, <C-unsigned-int>, <C-unsigned-short>, <DWORD>, <ULONG>, <USHORT>, <WORD>`

or any C pointer type. (But remember that arbitrary pointers can generally be meaningfully used only with an in-process server.)

Note that because of current implementation limitations, the type name must literally be one of those listed above, not some other name that is defined to be equal to one of them.

See also the **`define vtable-interface`** macro, which is like **`define custom-interface`** except that it also generates type information for creating a type library.

Example

define custom-interface does not actually define a server implementation class; that must be done separately. For example:

```
define custom-interface <IWhatever> (<IUnknown>)
  // interface
  uuid "{DC86922A-16C3-11D2-9A67-006097C90313}";
  property foo :: <integer>;
  function bar (a :: <integer>)
    => (b :: <string>);
end custom-interface <IWhatever>;

define COM-interface <my-whatever> (<IWhatever>)
  // implementation
  slot foo :: <integer>;
end COM-interface <my-whatever>;

define method bar (this :: <my-whatever>,
                   a :: <integer>)
    => (result :: <SCODE>, b :: <string>)
  …
  values($S-OK, ...)
end method bar;
```

The above defines a class **<IWhatever>** to represent a new COM interface, and a class **<my-whatever>** that provides an implementation of that interface.

define coclass — *Definition macro*

Summary

Creates type information for a COM class (coclass) implemented in Dylan.

Macro call

define coclass ***coclass-name***
  { ***typelib-clause*** | ***interface-clause*** }*
end [**coclass**] [***coclass-name***]

Arguments

coclass-name name_{bnf}

typelib-clause

```
{ uuid uuid; | name name; | documentation documentation;
| help-file help-file | help-context help-context;
| major-version major-version;
| minor-version minor-version;
| locale locale;
| component-class component-class;
| args ( keyword-arg-list );
}
```

keyword-arg-list

```
{ arg-keyword arg-value, }*
```

interface-clause

```
[ default | source | restricted ] interface
  interface-name
    { , interface-clause-option }* ;
```

interface-clause-option

```
{ name: name | typeinfo: typeinfo |
  args: { sequence | { keyword-arg-list } }
}
```

Description

Creates type information for a COM class (coclass) to implement a set of interfaces. *coclass-name* is bound to an instance of **`<coclass-type-info>`**.

Any *typelib-clause*s are passed on as keywords to the call to **`make(<coclass-type-info>)`** which is used to create the type information. The **`uuid`** and **`name`** clauses are required. The **`uuid`** clause accepts instances of **`<string>`** and **`<REFGUID>`**. The **`name`** clause is a Dylan name_{bnf} (as defined in the DRM's Dylan grammar). The **`component-class`** clause corresponds to the **`class:`** keyword; **`class`** could not be used as a clause name because of restrictions in the Dylan macro system.

When a client initiates a server for the class ID specified in the **uuid** clause, an instance of the class specified in the **component-class** clause will be created, using any additional **make** arguments specified by the **args** clause.

interface-clause declarations include interface implementations in the coclass. Each *interface-name* is the class that should be instantiated to implement the interface; usually it will have been defined with one of the macros **define dispatch-interface**, **define dual-interface**, or **define vtable-interface**.

One of the interfaces can be designated as the **default** interface—this is the one which is returned from a call to **create-dispatch** on the coclass (unless another interface is requested in the call). If no interface is designated as the **default** then the first interface mentioned is considered the default. **restricted** interfaces cannot be accessed by certain tools—Visual Basic, for example. **source** interfaces are interfaces that the client can implement in order to receive notifications from the server.

The *interface-clause-option* specifies additional options for making a **<component-interface-description>** which specifies how to instantiate the interface. Most of these options are unnecessary when the information has been provided in the **define dispatch-interface** macro, but the **args:** option can be used to specify additional keyword arguments for making the interface class.

Example

Here is an example of typical usage of **define coclass**:

```
define dispatch-interface <disp-interface-1>

  (<simple-dispatch>)
  …
end dispatch-interface <disp-interface-1>;
define dispatch-interface <disp-interface-2>
  (<simple-dispatch>)
  …
end dispatch-interface <disp-interface-2>;
```

```
define coclass $coclass-type-info
  name "coclass-type-info";
  uuid "{94CCDC4B-92AD-11D1-9A5E-006097C90313}";
  default interface <disp-interface-1>;
  interface <disp-interface-2>;
end coclass $coclass-type-info;

let factory :: <class-factory> =
  make-object-factory($coclass-type-info);
```

The above binds coclass type information to **`$coclass-type-info`**, and creates a class factory that will create instances of the coclass as necessary.

define vtable-interface *Definition macro*

Summary
Creates a new custom v-table COM interface and associated type information for a type library.

Macro call
define [***class-adjectives***] **vtable-interface** ***interface-name***
 (*superclass*)
  [**client-class** ***client-class-name***
  [:: ***client-superclass-name***] ;]
  { ***typelib-clause*** | ***property*** | ***function*** }*
end [**vtable-interface**] [***interface-name***]

Arguments

class-adjectives	Any Dylan class definition adjective.
interface-name	name_{bnf}
superclass	name_{bnf}
client-class-name	name_{bnf}

typelib-clause

```
{ uuid uuid;
| name name;
| documentation documentation;
| help-file help-file;
| help-context help-context;
| major-version major-version;
| minor-version minor-version;
| locale locale;
}
```

property

```
{ [ constant ] property property-name :: property-type
      { , option }* ;
}
```

function

```
{ function | vtable-member } member-name
    ( { argument-name :: argument-type }* )
      => ( { result-name :: result-type }* )
    { , option }* ;
```

option

```
{ name: name
| documentation: documentation
| help-context: help-context
}
```

Description

Used to create a new custom v-table COM interface and the associated type information for a type library. It combines the functionality of **define custom-interface** and **define vtable-type-info** such that the defined class can be used to describe an interface in the **define coclass** macro.

It defines the v-table and associated methods, but does not actually define a server implementation class; that must be done as a separate subclass (see example).

The syntax and usage are the same as for the **define custom-interface** macro, except that this macro also generates type information and supports some additional options for that purpose.

The *superclass* specifies the interface class to inherit from. Often it will be **<IUnknown>**, but it could be **<IDispatch>** or another class defined by **define vtable-interface**.

An instance of **<vtable-type-info>** (a subclass of **<ITypeInfo>**) is created to represent the interface to the class. This value can be fetched by applying the function **type-information** to the class or to an instance of the class. Any *typelib-clause*s are passed on as keywords to the call to **make(<vtable-type-info>),** which is used to create the type information.

Example

```
define vtable-interface <IWhatever> (<IUnknown>)
  // interface
  uuid "{DC86922A-16C3-11D2-9A67-006097C90313}";
  name "IWhatever";
  documentation "simple example of v-table interface";
  property foo :: <integer>;
  function bar (a :: <integer>)
    => (b :: <string>);
end vtable-interface <IWhatever>;

define COM-interface <my-whatever> (<IWhatever>)
  // implementation
  slot foo :: <integer>;
end COM-interface <my-whatever>;

define method bar (this :: <my-whatever>,
                   a ::   <integer>)
    => (result :: <SCODE>, b :: <string>)
  …
  values($S-OK, ...)
end method bar;
```

The above defines a class **<IWhatever>** to represent a new COM interface, and creates an **ITypeInfo** interface that can be accessed by calling the function **type-information** on the class or an instance.

define dispatch-client *Definition macro*

Summary Generates simple Dylan clients to COM dispatch interfaces.

Macro call

```
define dispatch-client interface-name
  { uuid | property | function }*
end [dispatch-client] [interface-name]
```

Arguments

interface-name $name_{bnf}$

uuid A GUID that identifies the interface.

The default is **$IID-IDispatch**.

This value should preferably be an instance of **<REFGUID>** (from **make-GUID**) but can also be a string representation of the GUID (containing 32 hexadecimal digits within braces).

property

```
[ size | element ] [ constant | write-only ] property
  prop-name
        [ ( { argument-name [ :: argument-type ]}* ) ]
        [ :: property-type ]
        { , option }* ;
```

function

```
function member-name
      [ ( { argument-name [ :: argument-type ]}* )
        [ => ( { result-name [ :: result-type ]}* ) ]
      ] { , option }* ;
```

option

```
{ name: name | disp-id: disp-id }
```

Description Generates simple Dylan clients to COM dispatch interfaces. It defines a class *interface-name* to represent the COM interface described. The class is a subclass of **<dispatch-client>**, and of **<collection>** if the **size** or **element** adjectives are present.

The **<dispatch-client>** class is a subclass of **<LPDISPATCH>**, so instances of dispatch-client types can be passed directly to functions such as **AddRef** and **Release**.

If a *uuid* clause is present, it specifies the interface ID for the interface. This value can be retrieved by applying the method **dispatch-client-uuid** to either the **<dispatch-client>** class or an instance.

A **make** method is defined on *interface-name*. No matter how the interface is created, it should be released with **Release** when it is no longer needed. The **make** method can be invoked with either a **disp-interface:** keyword or a **class-id:** keyword, and, optionally, with **interface-ID:** keyword. The reference information for these keywords is as follows:

disp-interface:

Specifies an **<LPDISPATCH>** or a **<dispatch-client>** to create the new interface from. If an interface ID is provided via the **uuid** clause or via the **interface-ID:** keyword, **QueryInterface** is used to find the new interface, otherwise the existing interface is used (after having **AddRef** applied).

class-id:

The UUID of a coclass which implements the desired interface. The coclass is created with **create-dispatch**. If an interface ID is provided via the **uuid** clause or via the **interface-ID:** keyword, that interface ID is requested, otherwise **$IID-IDispatch** is requested.

`interface-id:` (optional)

If a **`<string>`** or **`<LPGUID>`** is provided, that GUID is used as an interface ID rather than any provided by the **`uuid`** clause. If **`#f`** is provided for the **`interface-ID:`** keyword, then no interface ID (when **`disp-interface:`** is provided) or **`$IID-IDispatch`** (when **`class-id:`** is provided) is used.

For every *property* or *function* a getter named *member-name*-**`dispid`** is defined. This getter, when passed an instance of *interface-name*, returns the dispatch ID for *member-name*. If a value was provided by the **`disp-id:`** *option*, that value is returned, otherwise the dispatch ID is looked up with **`get-id-of-name`** and cached in a private per-instance slot. The name that is looked up is that specified by the **`name:`** *option*, or if not supplied the name of the property / method.

property declarations cause the appropriate getter and setter methods to be defined. **`constant`** properties have only a getter; **`write-only`** properties have only a setter; other properties have both. The getter method for a property accepts the **`default:`** keyword. If the server does not return a value (common for indexed properties), the value of **`default:`** is returned. If **`default:`** is not specified or **`$unsupplied`** and the server does not return a value, an error is signalled.

If a property has arguments, the setter / getter methods accept those arguments and treat the property as an OLE indexed property. An indexed property with a single argument may have the **`element`** adjective applied.

If a property has the **`element`** adjective applied then **`element`** getter and / or setter methods are defined and the class inherits from **`<collection>`**. The **`element`** getter / setter methods simply call the setters / getters for the property. For the **`element`** adjective to be applied to a property, the property must take a single argument and return a single value.

If a property has the **size** adjective applied then **size** getter and / or setter methods are defined and the class inherits from **<collection>**. The **size** getter / setter methods simply call the setters / getters for the property. For the **size** adjective to be applied to a property, the property must not take any arguments and must return an integer.

function declarations cause a method to be defined. The method accepts an instance of *interface-name* and any additional arguments specified in the declaration. The method simply invokes **call-simple-method** with the appropriate dispatch ID. Parameters and return values that are subtypes of **<dispatch-client>** are automatically translated to / from **<LPDISPATCH>**.

Example

An invocation of the macro looks much like a **define class** or a **define dispatch-interface** invocation:

```
define dispatch-client <IBlorf>
  uuid "{1f5bfc72-fa7a-11d1-a3c3-0060b0572a7f}";
  property IBlorf/Foo :: <integer>, disp-id: 7;
  function IBlorf/Bar (a :: <integer>)
    => (b :: <string>),
    name: "mBar";
end dispatch-client <IBlorf>;
```

The above defines a class **<IBlorf>** to represent the COM interface described, and defines methods **IBlorf/Foo**, **IBlorf/Foo-setter**, and **IBlorf/Bar** which call **call-simple-method** with the appropriate DISPID.

define dispatch-interface *Definition macro*

Summary

Provides a simple way to implement a server for a COM class with dispatch via **IDispatch**.

Macro call

define dispatch-interface ***interface-name*** **(*superclasses*)**
{ ***typelib-clause*** | ***property*** | ***function*** }*
end [**dispatch-interface**] [***interface-name***]

Arguments

interface-name $name_{bnf}$

superclasses $name_{bnf}$

interface-name $name_{bnf}$

typelib-clause

```
{ uuid string-or-REFGUID;
| name name;
| documentation documentation;
| help-file help-file;
| help-context help-context;
| major-version major-version;
| minor-version minor-version;
| locale locale;
}
```

property

```
{ slot-adjectives slot slot-name slot-stuff;
| constant property property-name [ :: property-type ]
    [ = property-value ] { , option }* ;
| [ virtual ] property property-name [ :: property-type ]
{ , option
| , slot-option }* ;
}
```

function

```
function member-name
  [ ( { argument-name :: argument-type ,}* )
    [ => ( [ result-name :: result-type ] ) ]
  ] { , option }* ;
```

option

```
{ name: name
| disp-id: disp-id
| documentation: documentation
| help-context: help-context
}
```

Description

Used to create Dylan COM servers with dispatch interfaces. By using **define COM-interface**, a Dylan class *interface-name* is created to implement the server.

The *superclasses* list specifies the classes to inherit from. Either **`<simple-dispatch>`** or another server class defined with **`define dispatch-interface`** must be included in the superclass list. If another server class is included, the type information for this class inherits the type information from the superclass.

An instance of **`<disp-type-info>`** is created to represent the interface to the class. This value can be fetched by applying the function **`type-information`** to the class or to an instance of the class. Any supplied **`typelib-clause`** clauses are passed on as keywords to the call to **`make(<disp-type-info>),`** which is used to create the type information.

property declarations correspond to slots in the class. **`slot`** defines a slot in the class (using the same syntax as slot declarations in class definitions), but does not create type information for it. Thus slots created with **`slot`** cannot be accessed by clients of the defined dispatch class. Slots defined with **`property`** do have corresponding information in the type library and can be accessed by clients. **`virtual`** properties only have type information, but no corresponding slot; getter and setter methods must be defined separately.

A constant **`property`** has no setter method, and must specify either a *property-value* or *property-type* or both. If a value is specified, then nothing further is needed to implement the property. Otherwise, a getter method will need to be defined separately.

The *property* clause has two extensions. The first extension allows a type declaration and an initialization expression, as in:

```
property foo :: <bar> = baz(), ...;
```

The type and the init expression become the type and init of the underlying Dylan slot, and also become the defaults for the **`type:`** and **`value:`** options of the property.

The second extension to *property* is an option **`property-setter`**, which can be used to specify a function to use to implement the setting of the property:

```
property foo, property-setter: external-foo-setter;
```

If **`property-setter:`** is not specified, then the **`slot`** setter is used.

function declarations cause a method description to be added to the type information for the class. The method must be defined separately. The method must accept an instance of *interface-name* and any additional arguments specified in the declaration. The method must return an **`<SCODE>`** instance describing the success or failure of the method, plus any other result specified (no more than one).

If the type of a property or a member function parameter list or result are not specified, they will be given default values from the definition of the implementing function, or the value of a constant property. C-FFI types may be used here in order to more precisely specify how the data will be represented in the interface.

While this macro is often used to directly implement a server, if you want to just define an abstract interface which could have multiple implementations, use only **`virtual property`** clauses instead of **`slot`** clauses, and define the slots in implementation subclasses defined by **`define COM-interface`**.

Example

An invocation of the macro looks much like a **`Define class`** or a **`define dispatch-client`** invocation:

```
define dispatch-interface <blorf-server>
 (<simple-dispatch>)
  uuid "{26d99e70-07ca-11d2-a3c5-0060b0572a7f}";
  property foo :: <integer>,
                  disp-id: 7;
  function bar (a :: <integer>)
    => (b :: <string>),
                  name: "mBar";
end dispatch-interface <blorf-server>;

define method bar (this :: <blorf-server>,
                   a :: <integer>)
    => (result :: <SCODE>, b :: <string>)
  …
  values($S-OK, ...)
end method bar;
```

The above defines a class **<blorf-server>** to implement the COM interface described, and creates type information describing the interface.

define dual-interface *Definition macro*

Summary: Creates a new dual COM interface.

Macro call:

define [***class-adjectives***] **dual-interface** ***interface-name***
 (*superclass*)
  [**client-class** ***vtable-client-class-name***
  [**::** ***client-superclass-name***] **;**]
  { ***typelib-clause***
  | ***property***
  | ***function*** }*
end [**dual-interface**] [***interface-name***]

Arguments:

class-adjectives Any Dylan class definition adjective.

interface-name name_{bnf}

vtable-client-class-name

name_{bnf}

typelib-clause

```
{ uuid uuid;
| name name;
| documentation documentation;
| help-file help-file;
| help-context help-context;
| major-version major-version;
| minor-version minor-version;
| locale locale;
}
```

property

```
{ [ virtual ] property property-name :: property-type
        { , option }* ;
| constant property property-name :: property-type
         [ = property-value ] { , option }* ;
}
```

function

```
function member-name
  ( { argument-name :: argument-type, }* )
      => ( [ result-name :: result-type ] )
  { , option }* ;
```

option

```
{ name: name
| disp-id: disp-id
| documentation: documentation
| help-context:  help-context
}
```

Description

Used to create a dual interface, with the members being accessible either through a v-table or through **`IDispatch/Invoke`**. This macro combines the functionality of **`define vtable-interface`** and **`define dispatch-interface`**. Typically it is used to define a class that just specifies the interface, with implementations of the interface being defined by subclasses.

The *superclass* list specifies the class to inherit from. It should be either **`<simple-dispatch>`** or another server class defined with **`define dual-interface`** or **`define dispatch-interface`**.

If another server class is included, the type information for this class inherits the type information from the superclass.

An instance of **`<dual-type-info>`** (a subclass of **`<vtable-type-info>`**) is created which is an **`ITypeInfo`** interface describing the dual interface.

This value can be fetched by applying the function **`type-information`** to the class or to an instance of the class.

Call function **`dispatch-type-information`** to obtain the instance of **`<disp-type-info>`** instead.

The clauses of the definition have the same meaning as documented for the **`define vtable-interface`** and **`define dispatch-interface`** macros. However, compared to **`define vtable-interface`**, dispatch interfaces impose several restrictions:

- The **`vtable-member`** clause cannot be used. (All member functions will implicitly return a status code.)
- No more than one result value can be returned in addition to the implicit status code.
- The data types for properties, parameters, and return values are restricted to those supported by OLE Automation. (Refer to section 4.8 of the *OLE, COM, ActiveX and DBMS* library reference.)

  Compared to **`define dispatch-interface`**, dual interfaces impose the following restrictions:

- The **`uuid`** clause is required.
- The **`slot`** clause cannot be used; slots should be defined in a separate implementation subclass.

- Properties should be declared as **`virtual`** in an interface class that will use a separate implementation subclass.
- Property types and function parameter and result lists must be explicitly declared instead of defaulting from the function definition. (This implementation limitation is because the types are needed for compile-time generation of v-table functions, not just for the run-time creation of type info data.)
- Do not call **`exit-invoke`** or **`abort`** in the implementation method for a dual interface member function or property because there will be no condition handler when it is invoked through the **`v-table`**.

Example

```
define dual-interface <IDual> (<simple-dispatch>)
  client-class <C-IDual>;
  uuid "{DC86922A-16C3-11D2-9A67-006097C90313}";
  virtual property foo :: <integer>;
  function bar (a :: <integer>)
    => (b :: <string>);
end dual-interface <IDual>;

define COM-interface <my-dual> (<IDual>)
  slot foo :: <integer>;
end COM-interface <my-dual>;

define method bar (this :: <my-dual>, a :: <integer>)
               => (result :: <SCODE>, b :: <string>)
  …
  values($S-OK, ...)
end method bar;
```

The above defines a class **`<IDual>`** which specifies a dual interface, and a subclass **`<my-dual>`** which implements a server for it. It also defines a class **`<C-IDual>`** for v-table client access, and creates type information describing the interface.

6

The OLE-Server Library

6.1 Introduction

This chapter describes the facilities of the OLE-Server library. You should use this library if you want to write an OLE compound document server application but you do *not* want to use the DUIM library to define its graphical user interface. Otherwise, the DUIM-OLE-Server library offers a simpler means of implementing compound document servers. See Chapter 3, “Compound Documents and OLE Controls in DUIM” for more information.

6.2 About compound documents

Compound documents are basically documents that contain data from more than one application. When viewing the document, you can see and edit the data from both applications.

For example, a compound document could be a text document from a word processor that contains a picture from a painting application. In OLE/COM terminology, the word processor is the *container application* and the painting application is the *server application*.

6.3 Basics of writing OLE compound document servers

To write an OLE compound document server application, use the OLE-Server library and module. Note that to implement the user interface for your server, you will also have to use Harlequin Dylan's Win32 API libraries; see the manual *C FFI and Win32 Reference* for details.

Your compound document server can be a *local server* (a server that is built as a .EXE file and therefore runs in its own process) or an *in-process* server (a server that is built as a .DLL file and that runs in its container's process). There is no direct support for remote servers.

Compound document servers are required to support several standard COM interfaces. Rather than requiring you to write support for each interface separately, Harlequin Dylan provides "framework" classes that encapsulate the necessary COM interfaces. To implement a server you define a subclass of the appropriate framework class, and then define various required and optional Dylan methods on it.

For local servers, you must define a subclass of **`<ole-server-framework>`**, page 243; define a subclass of **`<ole-in-process-server>`**, page 244, for in-process servers. The required Dylan methods are described in Section 6.7.2 on page 245 while the optional ones are described in Section 6.7.5 on page 257.

6.4 Example

This chapter provides more of a brief outline of OLE server implementation than a comprehensive specification. Many of the details will be clearer by examining the accompanying example program, available from the Harlequin Dylan Examples dialog as the "Win32 OLE Server" example under "OLE".

You can find the source folder for this example, which includes a README, under the Harlequin Dylan installation folder (usually **`C:\Program Files\Harlequin\Dylan\`**) as

```
Examples\ole\win32-ole-server\
```

6.5 Implementing local servers

To implement a local server, the basic structure of your application should be the same as any GUI application using the Microsoft Win32 API, with the following exceptions.

- Define a subclass of **`<ole-server-framework>`** and implement the required Dylan methods described in Section 6.7.2 on page 245.

- Make your application capable of self-registration with the Windows Registry. With self-registration, the application can record itself in the Registry as a server for the COM objects it implements. The OLE-Server contains utilities that make writing a self-registering local server application simpler.

  Wrap the body of the server application with a call to the function **`OLE-util-register-only?`**, page 253, which will test whether the application was passed arguments requiring it to perform a registration operation. Depending on the result of the call, the application can either run as normal, or perform the requested registration operation and exit.

  ```
  if (OLE-util-register-only?()) // just [un]register & terminate
   register-ole-server(class-id, prog-id, title-string,
                       short-name: short-name-string);
  else // actually run the program
  …
  end if;
  ```

  When the application is wrapped in this code, if it is run with the **`/RegServer`** option (typically done as part of an installation script), it will just register itself and terminate, or if run with the **`/UnregServer`** option, it will try to unregister itself and terminate. Otherwise, **`OLE-util-register-only?`** returns **`#f`** and the application continues to run normally.

- Once it is past the self-registration wrapper, make your application call **`OLE-util-started-by-OLE?`**, page 281 to find out if it has been invoked by an OLE container. If the function returns **`#f`**, your application should just run as an independent application. If the function returns **`#t`**, your application should not show any of its windows; support code in the OLE-Server library will take care of that at the appropriate time.

- If your application is running under OLE, then before it enters its event loop, it must make an instance of the Dylan class **`<class-factory>`**, page 286, passing it the application's COM Class ID and the Dylan object class to be instantiated — that is, your subclass of **`<ole-server-framework>`**.

  COM's class factories are a means of creating multiple instances of a particular COM class. Instantiating the class factory takes care of registering your server application as the process that will serve objects of the given COM class, so that the container can connect to it.

  (This registration should not be confused with the registration procedure discussed earlier, which was for registering the application as a server for the COM class or classes it supports, so that the container knows which server application to run.)

- When a container application connects to your server application, a single instance of your subclass is instantiated automatically by the class factory. The container will call **`QueryInterface`** on it to find the interfaces it needs. This object is the "controlling unknown" for all the low-level OLE interfaces; if necessary, your server could also call **`QueryInterface`** on it to obtain low-level interface pointers in order to extend the capabilities of the library.

- When your server application terminates, it should call **`revoke-registration`**, page 285, on the instance of **`<class-factory>`**.

- If your application is activated in-place, you must call the function **`OLE-util-translate-accelerator`** within the event loop to properly handle the top-level menu. It is harmless to call it in other contexts. Note that the COM interface object is created while events are being processed, so it cannot simply be placed in a local variable before entering the loop.

- If your application has created an interface object, then before it terminates (such as when responding to a **$WM-CLOSE** message), it must make sure that the interface is properly disconnected from the container. To do this, call the function **OLE-util-close-server** on the interface object.

  Because the OLE-Server instantiates the COM interface object for your server automatically, rather than through calling **make** on your subclass of **<ole-server-framework>**, your application will not naturally have a name bound to the COM interface object upon which you can call **<class-factory>**. You must arrange to bind the instance to a name by defining an **initialize** method on your subclass of **<ole-server-framework>**.

- Your application must have at least two windows: a top-level frame window (which includes the border and title bar), and a single child window that occupies the client area within the border, that is referred to as the *document window*. The document window can be subdivided into other child windows, but there must be a single window handle representing the whole client region.

- In stand-alone execution, your server application should use **ShowWindow** to display the frame window, as any Windows application would. For out-of-place activation under OLE, your application's frame window is not shown until the support library calls your application's **OLE-part-open-out** method.

- For in-place activation — that is, activation as an embedded object — the OLE-Server library calls the application's **OLE-part-doc-window** method to get the document window. The library then takes care of displaying the document window as a child of the container's window, and also automatically places a "hatch" border around it. Although the frame window is not used during in-place activation, it must still exist, to serve as the initial parent when creating the document window.

- You must set up the code that actually draws to the document window by calling it from either of two contexts: in response to a **$WM-PAINT** message, or when the application's **OLE-part-draw-metafile** method is called to produce an image of the window that will be displayed by the container when the server is not active, and will be saved with the container's other data.

- If your application has state information other than its screen image, as it most likely will, it will need to provide methods for handling that data's persistent storage. Even if it does not have such information it must provide methods, though they can do nothing. See the entries for **`OLE-part-Create-Streams`**, **`OLE-part-Open-Streams`**, **`OLE-part-Release-Streams`**, **`OLE-part-Save-To-Storage`**, and **`OLE-part-Load-From-Storage`** in Section 6.7.2.

6.6 Implementing in-process servers

If you want to build your server application as an in-process server — a server that runs within the client application's process — there are a few differences from what you would do for a local server, and a few additional considerations.

- You must define a subclass of **`<ole-in-process-server>`**, page 244, instead of **`<ole-server-framework>`**.

  The **`<ole-in-process-server>`** class is itself a subclass of **`<ole-server-framework>`**, so in-process servers inherit all methods documented for **`<ole-server-framework>`**.

- You must build your in-process server application as a .DLL file instead of as a .EXE file.

  Since your server application is a dynamic link library instead of a complete program, there will be no main program. The event loop will be provided by the container application.

- You can handle persistent storage differently from the way local server storage is handled. Another technique is described in Section 6.7.9 on page 292.

- Drawing is handled a little differently. Instead of calling **`OLE-part-draw-metafile`**, the function **`OLE-part-draw`**, page 296, is called. This function takes one additional parameter, which is a rectangle specifying the size and position within which to draw.

- Since there is no main program loop in an in-process server, self-registration and creation of a class factory are handled by code generated by the macro **initialize-ole-server**, page 291.

Overall, an in-process server application will look something like this:

```
define constant $my-class-ID = make-GUID(...);

define COM-interface <my-object-class> (<ole-in-process-server>)
  …
end;

… // methods on <my-object-class>

initialize-ole-server(<my-object-class>,
                      $my-class-ID,
                      "my.prog.id",
                      "title of my application");
```

6.7 The OLE-SERVER module

This section contains reference entries for interfaces in the OLE-Server module.

Note: Because OLE-Server uses and then reexports many interfaces from Harlequin Dylan's low-level OLE/COM API libraries it is not practical to provide entries here for all of them. You should look at the **library.dylan** file in the **Sources\ole\ole-server** folder in your Harlequin Dylan installation for the complete picture.

6.7.1 Framework classes and definition macro

<ole-server-framework> *Open abstract class*

Summary The class of local server applications.

Superclasses **<IUnknown>**

Init-keywords None.

Description The class of local server applications. If you are writing a local compound document server application, it must be defined as a subclass of this class.

You must define your subclass of **<ole-server-framework>** using **define COM-interface** (below) rather than **define class**. The syntax and semantics of **define COM-interface** are exactly the same as those for **define class**, but for implementation reasons it is not presently possible to use **define class**.

<ole-in-process-server> *Open abstract class*

Summary The class of OLE in-process server applications.

Superclasses **<ole-server-framework>**

Init-keywords None.

Description The class of OLE in-process server applications. If you are writing an in-process compound document server application, it must be defined as a subclass of this class.

Because this class is a subclass of **<ole-server-framework>**, in-process servers inherit all methods defined for local servers.

You must define your subclass of **<ole-server-framework>** using **define COM-interface** (below) rather than **define class**. The syntax and semantics of **define COM-interface** are exactly the same as those for **define class**, but for implementation reasons it is not presently possible to use **define class**.

define COM-interface *Definition macro*

Description

Defines COM interface subclasses. The syntax and semantics of this macro are exactly the same as those for **`define class`**, but for implementation reasons it is not presently possible to use **`define class`**.

6.7.2 Required methods for compound document servers

You must provide a method on each of the following generic functions for your server application to work, whether it is an in-process server or a local server. The OLE-Server library will make calls to these methods.

In each case, the first argument (named *obj* here) is to be specialized on the application's subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

OLE-part-doc-window *G.f method*

Summary

Returns the handle of the application's document window.

Signature

`OLE-part-doc-window` ***obj*** **`=>`** ***doc-window***

Arguments

obj An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values

doc-window An instance of **`<HWND>`**.

Description

Returns the handle of the application's document window, an instance of **`<HWND>`**.

You must provide a method for this function, specialized on your application's subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**, because the OLE-Server library will call it.

If your method returns a null handle (for example, **$NULL-HWND**), then in-place activation of your server will not be supported, and it will be activated out of place instead. In this case, you must also supply a method for **OLE-part-requested-size**, page 260.

OLE-part-open-out *G.f. method*

Summary Do out-of-place activation by showing the frame window.

Signature **OLE-part-open-out *obj* => *frame-window***

Arguments *obj* An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

Values *frame-window* An instance of **<HWND>**.

Description Do out-of-place activation by showing the frame window. The value is the handle of the frame window, an instance of **<HWND>**.

You must provide a method for this function, specialized on your application's subclass of **<ole-server-framework>** or **<ole-in-process-server>**, because the OLE-Server library may call it.

If your method returns a null handle (for example **$NULL-HWND**), the container application is told that the activation has failed.

OLE-part-draw-metafile *G.f. method*

Summary Draw all of the document window display to a device context.

Signature **`OLE-part-draw-metafile`** ***obj hDC*** **`=>`** ***status***

Arguments

obj An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**

hDC An instance of **`<HDC>`**.

Values None.

Description Draw all of the document window display to the device context *hDC*. This image is what the container displays when the server is not active.

You must provide a method for this function, specialized on your application's subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**, because the OLE-Server library may call it.

Normally, your method should return **`$S-OK`**, but you can also return an error status code (instance of **`<HRESULT>`**) that will be reported back to the container application.

terminate *G.f. method*

Summary A hook method for cleanup events in your server application when the container disconnects.

Signature **`terminate`** ***obj*** **`=> ()`**

Arguments

obj An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values None.

Description

A hook method for cleanup events in your server application. The method is called by the OLE-Server library when the container disconnects.

Your method must take whatever action is appropriate given that the object *obj* is no longer in use. Typically, your method should post a close message to cause the process to be terminated. For example:

```
define method terminate (obj :: <my-server-obj>) => ()
  next-method();
  if (OLE-util-started-by-OLE?()) // if started by OLE
    // post close message to terminate the process.
    PostMessage(obj.main-window, $WM-SYSCOMMAND,
                $SC-CLOSE, 0);
  end if;
  values()
end;
```

Note: Your method must always call **next-method** first, because there is a default method that does important bookkeeping.

6.7.3 More required methods (persistent storage)

The following group of methods implement the persistent storage of any server data that you need to save in addition to its window image. If there is no such data, you can define these methods to do nothing; but you must be sure to define them even so, because there are no default methods.

We have not supplied default methods because most applications will need persistent storage, and to allow these operations to be silently ignored could lead to difficulties that would be difficult to trace.

OLE-part-Create-Streams *G.f. method*

Summary

Create any COM **IStream** instances needed for persistently storing server data other than its window image.

Signature

OLE-part-Create-Streams ***obj storage*** **=> ()**

Arguments

obj An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

storage An instance of **<LPSTORAGE>**.

Values

None.

Description

Create any COM **IStream** instances needed for persistently storing server data other than its window image. The streams are ways of accessing a portion of data within the supplied *storage* object. An implementation for this method is mandatory; see below.

The usual implementation of this method will be like this:

```
obj.my-stream :=
  OLE-util-Create-Stream(storage, stream-name);
```

for each stream needed, where the second argument, *stream-name*, is an OLE string (an instance of **<LPOLESTR>**) that is the name of the stream. For example:

```
define constant $my-stream-name =
  as(<LPOLESTR>, "MyData");
```

If your server application has no data other than its window image to store persistently, your method can do nothing. But you must define one even so, because there is no default method. We have not supplied default methods because most applications will need persistent storage, and to allow this operation to be silently ignored could lead to difficulties that would be difficult to trace.

OLE-part-Open-Streams *G.f. method*

Summary

Open any COM **IStream** streams used for persistently storing server data other than its window image.

Signature

OLE-part-Open-Streams ***obj storage*** **=> ()**

Arguments

obj An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

storage An instance of **<LPSTORAGE>**.

Values

None.

Description

Open any COM **IStream** streams used for persistently storing server data other than its window image.

The usual implementation of this method will be like this:

```
obj.my-stream :=
  OLE-util-open-stream(storage, stream-name);
```

for each stream needed, where the second argument, *stream-name*, is an OLE string (an instance of **<LPOLESTR>**) that is the name of the stream.

If your server application requires persistent storage of no other data than its window image, your method can do nothing. But you must define one even so, because there is no default method. We have not supplied default methods because most applications will need persistent storage, and to allow this operation to be silently ignored could lead to difficulties that would be difficult to trace.

OLE-part-Release-Streams *G.f method*

Summary

Call the COM method **Release** (from **IUnknown**) on each of the COM **IStream** objects your server has open, and forget them.

Signature

OLE-part-Release-Streams ***obj*** **=> ()**

Arguments

obj An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

Values None.

Description Call the COM method **`Release`** (from **`IUnknown`**) on each of the COM **`IStream`** objects your server has open, and forget them.

If your server application requires persistent storage of no other data than its window image, your method can do nothing. But you must define one even so, because there is no default method. We have not supplied default methods because most applications will need persistent storage, and to allow this operation to be silently ignored could lead to difficulties that would be difficult to trace.

OLE-part-Save-To-Storage *G.f. method*

Summary Saves your server application's non-screen-image persistent data.

Signature **`OLE-part-Save-To-Storage`** ***obj storage sameasload?*** **`=> ()`**

Arguments

obj	An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.
storage	An instance of **`<LPSTORAGE>`**.
sameasload?	An instance of **`<boolean>`**.

Values None.

Description Saves your server application's non-screen-image persistent data to *storage* by writing it to the application's stream objects.

The *sameasload?* flag indicates whether this is the same compound document storage as that with which the object was originally created. If so, your method should write to the same streams remembered from a previous call to **`OLE-part-Create-Streams`** or **`OLE-part-Open-Streams`**; otherwise, **`OLE-part-Create-Streams`** should be called to create new temporary streams (which should be released with **`Release`**, page 317, before returning).

You can call **`istream-rewrite`** on each stream to position it at the beginning and erase any old data, or use **`IStream/Seek`** to do selective updates. The writing can be done either by using helper functions such as **`istream-write-integer`** and **`istream-write-int16`**, or by using the low-level API function **`IStream/Write`**.

If your server application has no data other than its window image to store persistently, your method can do nothing. But you must define one even so, because there is no default method. We have not supplied default methods because most applications will need persistent storage, and to allow these operations to be silently ignored could lead to difficulties that would be difficult to trace.

OLE-part-Load-From-Storage *G.f. method*

Summary Initializes the size of your server application's document window to given dimensions, and loads the application's persistently stored non-screen-image data.

Signature **`OLE-part-Load-From-Storage`** ***obj width height*** **`=> ()`**

Arguments

obj	An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.
width	An instance of **`<integer>`**.
height	An instance of **`<integer>`**.

Values None.

Description

Initializes the size of your server application's document window to the saved *width* and *height* (in integer pixel units), and loads the application's persistent data from storage by reading from your stream objects (which were created by **`OLE-part-Open-Streams`**).

The reading can be done either by using helper functions such as **`istream-read-integer`** and **`istream-read-int16`**, or by using the low-level API function **`IStream/Read`**.

If your server application has no data other than its window image to store persistently, your method can do nothing. But you must define one even so, because there is no default method. We have not supplied default methods because most applications will need persistent storage, and to allow these operations to be silently ignored could lead to difficulties that would be difficult to trace.

6.7.4 Self-registration for local server applications

The following utilities allow you to arrange self-registration for local server applications.

OLE-util-register-only? *Function*

Summary

Returns **`#t`** if a local server application has been invoked with command-line arguments specifying that it only register or unregister itself, rather than run normally; and **`#f`** otherwise.

Signature

`OLE-util-register-only? () =>` ***register?***

Arguments

None.

Values

register? An instance of **`<boolean>`**.

Description

Returns **#t** if the application has been invoked with command-line arguments specifying only that it register or unregister itself, rather than run normally; and **#f** otherwise.

The function will return **#t** when the application's first command-line argument is any of **/RegServer**, **-RegServer**, **/UnregServer** or **-UnregServer**. The comparison of the actual command-line arguments with these strings is not case-sensitive.

Only local server applications should call this function; it is meaningless to call it in an in-process server application, since they cannot be invoked as stand-alone applications with command-line arguments.

If the result of calling this function is **#t**, the local server application should simply call **register-ole-server** and terminate without doing anything else. That function will register or unregister the application as the command-line arguments require.

Example

```
if (OLE-util-register-only?()) // just [un]register &
                               // terminate
 register-ole-server(class-id, prog-id, title-string,
                    short-name: short-name-string);
else // actually run the program
…
end if;
```

register-ole-server *Function*

Summary

Depending on the command-line arguments passed to the application, this function creates or deletes the Windows 95 and Windows NT system registry entries necessary for the current application to be used as a server for an embeddable OLE object.

Signature **`register-ole-server`** ***class-id prog-id title-string***
`#key` ***full-name short-name app-name***
misc-status verbs **`=> ()`**

Arguments

class-id — The COM Class ID that identifies the object the server provides. Required.

This ID can be represented either as a string of 32 hexadecimal digits in the following format

"{*xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*}"

That is, where each *x* is a hexadecimal digit. For example:

"{e90f09e0-43db-11d0-8a04-02070119f639}"

Alternatively, the ID can be an instance of **`<REFCLSID>`**, as returned from **`make-GUID`**.

prog-id — An instance of **`<string>`**. Required.

title-string — An instance of **`<string>`**. Required.

full-name — An instance of **`<string>`**. Default value: the value of *short-name*.

short-name — An instance of **`<string>`**. Default value: the value of *app-name*.

app-name — An instance of **`<string>`**. Default value: the value of *title-string*.

misc-status — An instance of **`<integer>`**.

verbs — An instance of **`<vector>`**. Default value: see Description.

Values None.

Description Creates the Windows 95 and Windows NT system registry entries necessary for the current application to be used as a server for an embeddable OLE object. But if the application was invoked with **/UnregServer** or **-UnregServer**, this function will try to delete any registry entries that belong to the application.

The *class-id* argument must have the same value as passed with the **clsid:** init-keyword to **make** on **<class-factory>**.

The *prog-id* argument is a unique internal string name for the class. It must begin with a letter. It cannot contain any spaces or punctuation except a period (.) character, and it must not be more than 39 characters long. In order to ensure uniqueness, the recommended format is:

vendor-name*.*product-name*.*version-number

For example:

```
AcmeWidgets.FrobMaster.2
```

The *title-string* argument appears in a container's Insert Object dialog box to identify the kind of object this server application provides.

The *full-name* keyword argument is the full type name of the class. Its default value is the value of the *short-name* keyword.

The *short-name* keyword argument is used for popup menus and the Links dialog box. It must be a string of not more than 15 characters in length. Its default value is the value of the *app-name* keyword.

The *app-name* keyword argument is the name of the application servicing the class, and is used in the Result text in dialog boxes. Its default value is the value of the required *title-string* argument.

The *misc-status* keyword argument specifies the value to be returned by **IOleObject::GetMiscStatus**. The value is formed by using **logior** to combine **$OLEMISC-**… constants. Default value: 0.

The *verbs* keyword argument specifies the verbs and menu entries that the object supports. The value is a vector, with each element representing the attributes of one verb. Default value:

```
vector(vector(0, "&Edit",  // in-place activation
              $MF-ENABLED,
              $OLEVERBATTRIB-ONCONTAINERMENU),
       vector(1, "&Open",  // out-of-place activation
              $MF-ENABLED,
              $OLEVERBATTRIB-ONCONTAINERMENU))
```

This value produces the following registry entries:

```
HKEY_CLASSES_ROOT\CLSID\{...}\Verb\0 = &Edit,0,2

HKEY_CLASSES_ROOT\CLSID\{...}\Verb\1 = &Open,0,2
```

Alternatively, you can register a server with:

```
// check for "/RegServer" or "/UnregServer"
if (~OLE-util-maybe-just-register(class-id,
                                  prog-id,
                                  title-string,
                                 short-name: name))
// now run the application
…
end if;
```

Here, the **OLE-util-maybe-just-register** function has the same arguments as **register-ole-server**. It either unregisters the server application and returns **#t**, or does nothing and returns **#f**.

6.7.5 Optional methods for compound document servers

You may wish to define some of the following methods. You are not required to provide an implementation; all have default methods.

OLE-part-init-new *G.f. method*

Summary
: Implement this method to carry out initialization events in your server; the OLE-Server library calls the method when a new object is being created.

Signature
: **`OLE-part-init-new`** ***obj*** **`=> ()`**

Arguments
: *obj* — An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values
: None.

Description
: Implement this method to carry out any desired initialization events in your server; the OLE-Server library calls the method when a new object is being created—that is, when **`OLE-part-Load-From-Storage`** is not going to be called to load an old object.

The default method does nothing.

OLE-part-dirty? *G.f. method*

Summary
: Implement this method to tell the OLE-Server whether your application's persistent non-screen-image data has changed since it was last loaded or stored.

Signature
: **`OLE-part-dirty?`** ***obj*** **`=>`** ***dirty?***

Arguments
: *obj* — An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values
: *dirty?* — An instance of **`<boolean>`**.

Description

Implement this method to tell the OLE-Server whether your application's persistent non-screen-image data has changed since it was last loaded or stored.

Your method should return true if the application's data has changed; otherwise false.

The OLE-Server library uses this method to decide whether it needs to call your **`OLE-part-Save-To-Storage`** method.

Typically you would want your object to contain a slot which this method would return, that is set to **`#f`** by **`OLE-part-Save-To-Storage`** and **`OLE-part-Load-From-Storage`**, and set to **`#t`** when any relevant data is changed.

The default method always returns **`#t`**.

OLE-part-title *G.f. method*

Summary

Implement this method to give a title to your embedded server object.

Signature

`OLE-part-title` ***obj*** **`=>`** ***title***

Arguments

obj An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values

title An instance of **`false-or(<string>)`**.

Description

Implement this method to give a title to your embedded server object. Your method should return either **`#f`** or a string identifying the server application.

If it returns **`#f`** or an empty string, the embedded part will have no title.

Typically, this would be the same as what would be shown in the server application's title bar during stand-alone execution. However, most containers do not use this information.

The default method returns no title.

OLE-part-set-focus *G.f method*

Summary
Implement this method to set the focus to the embedded object's document window.

Signature
`OLE-part-set-focus` ***obj*** **`=> ()`**

Arguments
obj An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values
None.

Description
Implement this method to set the focus to the embedded object's document window.

The default method does:

```
SetFocus(OLE-part-doc-window(obj));
```

That is usually the right thing to do, but some applications might need to provide an override on this method for performing additional bookkeeping.

OLE-part-requested-size *G.f. method*

Summary
Implement this method to define the dimensions of the display region you want your embedded server object to occupy in the container.

Signature
`OLE-part-requested-size` ***obj*** **`=>`** ***width height***

Arguments *obj* An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

Values *width* An instance of **<integer>**.

height An instance of **<integer>**.

Description Implement this method to define the dimensions of the display region you want your embedded server object to occupy in the container. The OLE-Server library calls this method to find out what dimensions you would like the embedded object display to have.

Your method should return two integers, representing the region's width and height in pixel units.

A container can usually make the requested region available, but if it cannot, it calls **OLE-part-change-size** to report the size of the region it did allocate.

The default method uses **GetClientRect** to get the initial size of the window returned by **OLE-part-doc-window**.

OLE-part-change-size *G.f method*

Summary If the container changes the size of the display region it has allocated for the embedded object, the OLE-Server library calls this method to allow your server to make any changes it desires to its visual representation, or to refuse the change if it can no longer support in-place activation in the changed region.

Signature **OLE-part-change-size** ***obj width height*** **=>** ***OK?***

Arguments *obj* An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

	width	An instance of **<integer>**.
	height	An instance of **<integer>**.
Values	*OK?*	An instance of **<boolean>**.

Description

If the container changes the size of the display region it has allocated for the embedded object, the OLE-Server library calls this method to allow your server to make any changes it desires to its visual representation, or to refuse the change if it can no longer support in-place activation in the changed region. The container may have to change the size of the region if the container user resizes the embedded object.

The *width* and *height* arguments are the pixel units defining the region's dimensions after the change.

This method will not be called as long as the space is the same as requested by **OLE-part-requested-size**.

Your method should return **#t** if the change is acceptable, or **#f** if in-place activation can no longer be supported in the given space.

The default method does nothing and returns **#t**. You should supply an override method if you want to respond to a change by altering the embedded object display, such as by scaling or scrolling, instead of merely letting it be clipped. Note that the actual change to the window size will be carried out separately by **OLE-part-position-window**, so your **OLE-part-change-size** method is not expected to do that.

OLE-part-position-window *G.f. method*

Summary

The OLE-Server library calls this method during in-place activation, to set the size and position of the embedded object's document window, as designated by the given rectangle with pixel coordinates.

Signature **OLE-part-position-window** ***obj rect repaint?*** **=> ()**

Arguments

obj An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

rect An instance of **<LPRECT>**.

repaint? An instance of **<boolean>**.

Values None.

Description The OLE-Server library calls this method during in-place activation, to set the size and position of the embedded object's document window, as designated by the given rectangle, *rect*, with pixel coordinates.

If *repaint?* is true, your method should redraw the document window.

Note that the window size given here is what is currently visible in the container after clipping, and not necessarily the full size of the allocated space (as from **OLE-part-change-size**).

The default method does this:

```
MoveWindow(OLE-part-doc-window(obj),
  rect.left-value, rect.top-value,
  rect.right-value - rect.left-value,
  rect.bottom-value - rect.top-value,
  repaint?);
```

As is documented for the **IOleInPlaceObject/SetObjectRects** operation that uses it, this method must not make calls to the Windows **PeekMessage** or **GetMessage** functions, or to a dialog box.

OLE-part-hide *G.f. method*

Summary
Removes your server application's frame window from the screen.

Signature
OLE-part-hide ***obj*** **=> ()**

Arguments
obj An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

Values
None.

Description
Removes your server application's frame window from the screen.

The OLE-Server library calls this method during out-of-place activation only. The default method does this:

ShowWindow(*window*, $SW-HIDE);

where *window* is the value returned by **OLE-part-open-out**.

OLE-part-in-place-activated *G.f. method*

Summary
A hook method for events that should take place in your server once the display is initialized in an in-place activation.

Signature
OLE-part-in-place-activated ***obj*** **=> ()**

Arguments
obj An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

Values
None.

Description
A hook method for events that should take place in your server once the display is initialized in an in-place activation.

The OLE-Server library calls this method when in-place activation begins, immediately after it has completed displaying the server application's document window, menus, and tool-bar in the container.

The purpose of the call to this method is simply to allow your application to take any action that might be necessary once the display part of in-place activation is complete. This action could be bookkeeping or a visual change. The default method does nothing.

OLE-part-UI-activated *G.f. method*

Summary
A hook method for events that should take place in your server once the frame-level user interface (such as the menus and tool bar) has been added to the display in an in-place activation.

Signature
`OLE-part-UI-activated` ***`obj`*** **`=> ()`**

Arguments
obj An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values
None.

Description
The OLE-Server library calls this method once it has added the frame-level user interface (such as the menus and tool bar) to the display in an in-place activation.

The purpose of the call to this method is simply to allow your application to take any action that might be necessary at this point. This action could be bookkeeping or a visual change. The default method does nothing.

Usually, the in-place activation of an embedded server causes both **`OLE-part-in-place-activated`** and **`OLE-part-UI-activated`** to be called. However, for an OLE control (see

Chapter 7) which can be one of several active at the same time, **`OLE-part-UI-activated`** will not be called since the frame-level user interface is not altered.

OLE-part-in-place-deactivated *G.f. method*

Summary A hook method for events that should take place in your server when an in-place activation ends, just before the container's user interface is restored.

Signature **`OLE-part-in-place-deactivated`** ***obj*** **`=> ()`**

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values None.

Description The OLE-Server library calls this method when in-place activation is ended, just before restoring the container's user interface. It simply allows the application to recognize that this is going to happen and to perform any desired book-keeping. The default method does nothing.

`OLE-util-set-status-text`, page 280, can be called here to leave a parting message.

OLE-part-UI-deactivated *G.f. method*

Summary Implement this method to carry out events that should take place in your server before the frame-level user interface (such as the menus and tool bar) is removed from the display in an in-place activation.

Signature **`OLE-part-UI-deactivated`** ***obj*** **`=> ()`**

Arguments	*obj*	An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

Values None.

Description The OLE-Server library calls this method before removing any frame-level user interface (such as the menus and tool bar) from the display in an in-place activation.

The default method does nothing.

For an embedded document server, this is usually followed by a call to **`OLE-part-in-place-deactivated`**. However, an OLE Control (see Chapter 7) can remain in-place active even though it is no longer UI active.

OLE-part-insert-menus *G.f. method*

Summary Implement this method to specify server menus for merging into the container's menu bar during in-place activation.

Signature **`OLE-part-insert-menus`** ***obj hmenu edit-pos object-pos help-pos*** **`=> (`*nedit*`,` *nobject*`,` *nhelp*`)`**

Arguments	*obj*	An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.
	hmenus	An instance of **<HMENU>**.
	edit-pos	An instance of **<integer>**.
	object-pos	An instance of **<integer>**.
	help-pos	An instance of **<integer>**.
Values	*nedit*	An instance of **<integer>**.
	nobject	An instance of **<integer>**.

nhelp An instance of **<integer>**.

Description

Implement this method to specify server menus for merging into the container's menu bar during in-place activation.

The *hmenu* argument is the shared menu bar with the container's menus already installed.

The *edit-pos*, *object-pos*, and *help-pos* arguments are integers each specifying the positions in the menu bar at which menus can be inserted for each of three groups, designated "Edit", "Object", and "Help" menus.

Your method must return the number of menus added in each group. The default method does nothing, returning **values(0, 0, 0)**.

You can insert menus by calling the Win32 function **InsertMenu**. For example, if there is just one menu to be inserted, the method body might look something like this:

```
InsertMenu(hmenu, edit-pos,
               logior($MF-BYPOSITION,$MF-POPUP),
               pointer-address(my-hMenu), TEXT("&Mine"));
values( 1, 0, 0 )
```

OLE-part-release-menu *G.f. method*

Summary

When an in-place activation is ended, this method is called on each of the menus that were inserted by **OLE-part-insert-menus**.

Signature

OLE-part-release-menu *obj hmenu* => ()

Arguments

obj An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

hmenu An instance of **<HMENU>**.

Values None.

Description When an in-place activation is ended, this method is called on each of the menus that were inserted by **`OLE-part-insert-menus`**.

This provides an opportunity for you to call **`DestroyMenu`** if appropriate. The default method does nothing.

OLE-part-command-window *G.f. method*

Summary Return the window handle to which command messages will be directed when any items from the inserted menus are invoked.

Signature **`OLE-part-command-window`** ***obj*** **`=>`** ***window*** **`:: <HWND>`**

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values *window* An instance of **`<HWND>`**.

Description Return the window handle to which command messages will be directed when any items from the inserted menus are invoked.

The default method uses the document window; you will need to provide an override method if the menu commands are to be processed by the frame window.

OLE-part-accelerators *G.f method*

Summary Returns the handle of the application's accelerator key table, or **`#f`** if the application does not use any accelerators.

Signature **`OLE-part-accelerators`** ***obj*** **`=>`** ***table***

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values *table* An instance of **`false-or(<HACCEL>)`**.

Description Returns the handle of the application's accelerator key table, or **`#f`** if the application does not use any accelerators. The default method returns **`#f`**.

This method is used during in-place activation of an in-process server to handle keystrokes received by the container. An accelerator handle is obtained from the Win32 function **`LoadAccelerators`** or **`CreateAcceleratorTable`**, but those should not be called in this method since it will be called for each keystroke. Instead, this method should simply be an accessor for a previously computed value. Note that this should be the same value as is passed to **`TranslateAccelerator`** in the server's event loop.

OLE-part-toolbar-window *G.f method*

Summary Returns the window handle of the application's tool bar.

Signature **`OLE-part-toolbar-window`** ***obj*** **`=>`** ***toolbar-window***

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values *toolbar-window* An instance of **`<HWND>`**.

Description Returns the window handle of the application's tool bar. If the handle is not null, the window will be displayed at the top of the container window during in-place activation. The height of the window should have been set as desired before this function returns. The width of the window will be automatically adjusted to fit the container. The default method returns **`$NULL-HWND`**.

OLE-part-get-data *G.f. method*

Summary
: Implements the COM **`IDataObject/GetData`** operation, storing a representation of the document window.

Signature
: **`OLE-part-get-data`** ***obj pformatetc pmedium*** **`=>`** ***status***

Arguments
: *obj* — An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.
: *pformatetc* — An instance of **`<LPFORMATETC>`**.
: *pmedium* — An instance of **`<LPSTGMEDIUM>`**.

Values
: *status* — An instance of **`<HRESULT>`**.

Description
: Implements the COM **`IDataObject/GetData`** operation by storing a representation of the document window in the given *pmedium* (an instance of **`<LPSTGMEDIUM>`**) according to the format specified by *pformatetc* (an instance of **`<LPFORMATETC>`**).

If *pmedium* is a null pointer, this method should not actually store any data, but just indicate by the returned status whether the format is supported (this case is used to implement **`IDataObject/QueryGetData`**).

The default method creates a “metafile” and calls **`OLE-part-draw-metafile`** to draw into it. (Both old style and enhanced metafiles are supported for compatibility with both 16-bit and 32-bit container applications.) You should only need to override this if you need to support some other data format or some representation other than the entire window (**`$DVASPECT-CONTENT`**).

See also the companion function **`OLE-part-formats-for-get`** below.

OLE-part-set-data *G.f method*

Summary Implements the COM **`IDataObject/SetData`** operation by storing the given data into the application.

Signature **`OLE-part-set-data`** ***obj pformatetc pmedium*** **`=>`** ***status***

Arguments

obj	An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.
pformatetc	An instance of **`<LPFORMATETC>`**.
pmedium	An instance of **`<LPSTGMEDIUM>`**.

Values

status	An instance of **`<HRESULT>`**.

Description Implements the COM **`IDataObject/SetData`** operation by storing the given data into the application. The default method simply returns an error status indicating that this operation is not supported, which is sufficient for many applications. Most compound document containers do not call this.

OLE-part-formats-for-get *G.f method*

Summary Used in the implementation of the COM method **`IDataObject/EnumFormatEtc`** when the direction is **`$DATADIR-GET`**.

Signature **`OLE-part-formats-for-get`** ***obj*** **`=>`** ***formats***

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values *formats* An instance of **`<list>`**.

Description Used in the implementation of the COM method **`IDataObject/EnumFormatEtc`** (not used by all containers) when the direction is **`$DATADIR-GET`**. It returns a list in which each element is an instance of **`<FORMATETC-info>`** which describes a supported data format for **`IDataObject/GetData`**. The default method returns

```
list(make(<FORMATETC-info>, format: $CF-METAFILEPICT,
          aspect: $DVASPECT-CONTENT,
          tymed: logior($TYMED-MFPICT, $TYMED-ENHMF)))
```

If you override the default method for **`OLE-part-get-data`** to support additional formats, then a corresponding override should be provided here also. To add one additional format, the method body might look like:

```
pair( make(<FORMATETC-info>, …), next-method() )
```

OLE-part-formats-for-set *Open generic function*

Summary Used in the implementation of the COM method **`IDataObject/EnumFormatEtc`** when the direction is **`$DATADIR-SET`**.

Signature **`OLE-part-formats-for-set`** ***obj*** **`=>`** ***formats***

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values *formats* An instance of **`<list>`**.

Description Used in the implementation of the COM method **`IDataObject/EnumFormatEtc`** when the direction is **`$DATADIR-SET`**. It returns a list in which each element is a description of a supported data format for **`IDataObject/SetData`**.

The default method returns an empty list. If you override the default method for **`OLE-part-set-data`**, a corresponding override should be provided for this method too.

OLE-part-enable-dialog *Open generic function*

Summary Server applications should provide a method on this function if they display any modeless dialog boxes.

Signature **`OLE-part-enable-dialog`** ***obj enable?*** **`=> ()`**

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

enable? An instance of **`<boolean>`**.

Values None.

Description Server applications should provide a method on this function if they display any modeless dialog boxes.

When *enable?* is false, any modeless dialog boxes currently being displayed should be disabled because the container is showing a modal dialog box. When *enable?* is true, any modeless dialog boxes should be re-enabled. The default method does nothing.

6.7.6 Miscellaneous OLE-SERVER module classes and functions

You may wish to use some of the following functions and classes exported by the OLE-Server module.

<storage-istream> *Abstract class*

Summary The class of all OLE streams.

Superclasses **<LPUNKNOWN>** and **<positionable-stream>**

Description The class of all OLE IStream interface objects. It supports both the OLE IStream interface and the Dylan Streams library unbuffered positionable character stream protocol. Instances of **<storage-istream>** are returned by **OLE-util-Create-Stream** and **OLE-util-open-stream**, and are given as arguments to **OLE-part-Load-From-Stream** and **OLE-part-Save-To-Stream**.

istream-rewrite *Function*

Signature **istream-rewrite *stream* => ()**

Arguments *stream* An instance of **<storage-istream>**.

Values None.

Description Rewind the OLE data stream to its beginning and set its size to 0 in preparation for writing new data.

istream-write-integer *Function*

Signature **istream-write-integer *stream* *value* => ()**

Arguments *stream* An instance of **<storage-istream>**.

value An instance of **<integer>** or **<machine-word>**.

Values None.

Description Write an integer to the OLE data stream, as 32 bits.

istream-write-int16 *Function*

Signature **istream-write-int16** ***stream integer*** **=> ()**

Arguments *stream* An instance of **<storage-istream>**.

value An instance of **<integer>**.

Values None.

Description Write an integer to the OLE data stream, as only 16 bits.

istream-write-float *Function*

Signature **istream-write-float** ***stream value*** **=> ()**

Arguments *stream* An instance of **<storage-istream>**.

value An instance of **<single-float>**.

Values None.

Description Write a floating-point value to an OLE data stream.

istream-write-string *Function*

Signature **`istream-write-string`** ***stream value*** **`=> ()`**

Arguments

stream An instance of **`<storage-istream>`**.

value An instance of **`<byte-string>`**.

Values None.

Description Write a string value to an OLE data stream.

istream-read-integer *Function*

Signature **`istream-read-integer`** ***stream*** **`=>`** ***value***

Arguments

stream An instance of **`<storage-istream>`**.

Values

value An instance of **`<integer>`** or **`<machine-word>`**.

Description Read a value written by **`istream-write-integer`**. This function returns an instance of **`<machine-word>`** if the value does not fit in an **`<integer>`**.

istream-read-int16 *Function*

Signature **`istream-read-int16`** ***stream*** **`=>`** ***value***

Arguments

stream An instance of **`<storage-istream>`**.

Values

value An instance of **`<integer>`**.

Description Read a value written by **`istream-write-int16`**.

istream-read-float *Function*

Signature **istream-read-float** ***stream*** **=>** ***value***

Arguments *stream* An instance of **<storage-istream>**.

Values *value* An instance of **<single-float>**.

Description Read a value written by **istream-write-float**.

istream-read-string *Function*

Signature **istream-read-string** ***stream*** **=>** ***value***

Arguments *stream* An instance of **<storage-istream>**.

Values *value* An instance of **<byte-string>**.

Description Read a value written by **istream-write-string**.

OLE-util-Create-Stream *Function*

Summary Create an OLE storage stream.

Signature **OLE-util-Create-Stream** ***storage*** ***name*** **#key** ***mode*** **=>** ***stream***

Arguments *storage* An instance of **<LPSTORAGE>**.

name An instance of **<LPOLESTR>**.

mode An instance of **<integer>**. Default value: See Description.

Values *stream* An instance of **<storage-istream>**.

Description Create an OLE storage stream.

If you are writing a compound document server and want to implement persistent storage to save data (other than the embedded part's image, which is handled automatically) you will find this function useful when writing the method on **OLE-part-Create-Streams**, page 248.

The *mode* keyword in this function defaults to:

```
logior($STGM-READWRITE, $STGM-SHARE-EXCLUSIVE,
       $STGM-CREATE)
```

For an explanation of those constants, see the Microsoft OLE/COM API documentation for the **IStorage** method **CreateStream**.

OLE-util-open-stream *Function*

Signature **OLE-util-open-stream** ***storage name*** **#key *mode* => *stream***

Arguments

storage	An instance of **<LPSTORAGE>**.
name	An instance of **<LPOLESTR>**.
mode	An instance of **<integer>**. Default value: See Description.

Values

stream	An instance of **<storage-istream>**.

Description Open an OLE storage stream.

If you are writing a compound document server and want to implement persistent storage to save data (other than the embedded part's image, which is handled automatically), you will find this function useful when writing the method on **OLE-part-Open-Streams**, page 249.

The *mode* keyword in this function defaults to:

```
logior($STGM-READWRITE,$STGM-SHARE-EXCLUSIVE)
```

OLE-util-set-status-text *Function*

Signature **OLE-util-set-status-text** ***obj text*** **=>** ***status***

Arguments

obj	An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.
text	An instance of **false-or(<string>)**.

Values

status	An instance of **<HRESULT>**.

Description

Display the given string in the container application's status bar, or clear the status bar if *text* is **#f** or the empty string.

Typically this would be used in a window callback function to respond to a **$WM-MENUSELECT** message by showing a description of the menu item. The returned value could be one of:

$S-OK	Whole text successfully displayed. Satisfies the predicate **SUCCEEDED?**.
$S-TRUNCATED	Displayed part of message too long to fit. Satisfies the predicate **SUCCEEDED?**.
$E-FAIL	Container does not support a status bar. Satisfies the predicate **FAILED?**.
$OLE-E-NOT-INPLACEACTIVE	Not running in-place active. Satisfies the predicate **FAILED?**.

A string of type **<LPOLESTR>** is preferred, but any other type of **<string>** will be automatically converted. Thus, if a literal is being used, you might want to call **OLESTR** on it to cache the conversion.

OLE-util-started-by-OLE? *Function*

Signature **`OLE-util-started-by-OLE? () =>`** ***started-by-ole?***

Arguments None.

Values *started-by-ole?* An instance of **`<boolean>`**.

Description Returns **`#t`** if the program was initiated by OLE; **`#f`** if running by itself. Only relevant to a local server, not an in-process server.

OLE-util-in-process-startup? *Function*

See full reference entry **`OLE-util-in-process-startup?`**, page 204.

OLE-util-container-name *Function*

Signature **`OLE-util-container-name`** ***obj*** **`=> (`*application*`,`** ***document*`)`**

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values *application* An instance of **`false-or(<string>)`**.

document An instance of **`false-or(<string>)`**.

Description Returns the name of the container application program (or **`#f`** if not running under OLE or if called too soon) and the name of the container document (or **`#f`** if not applicable).

This function will not return meaningful data until the server is activated, so it should typically be called from the **`OLE-part-open-out`** or **`OLE-part-insert-menus`** methods.

You may wish to use this function to determine how to modify the **File > Exit** menu item when your server is running in an out-of-place activation: in such a case, the menu item should read **File > Exit to** *application*.

OLE-util-translate-accelerator *Function*

Signature `OLE-util-translate-accelerator` ***obj-or-false msg*** **=>** ***handled?***

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**, or false.

msg An instance of **`<LPMSG>`**.

Values *handled?* An instance of **`<boolean>`**.

Description If the server application is running in an in-place activation, and the window message is a keypress which is an accelerator key for the container application, then this function posts the corresponding command to the container and returns **`#t`**. Otherwise, it returns **`#f`**, indicating that the message should be passed on to **`DispatchMessage`**. This includes the case where *obj* is #f, which means that the COM object has not been instantiated yet.

You should use this function in the server's event loop after handling the server's own accelerators, like this:

```
while ( GetMessage(msg, $NULL-HWND, 0, 0) )
  // Check the server's accelerator keys first:
  if (zero?
       (TranslateAccelerator(msg.hwnd-value,
                             *my-accelerator-table*,
                             msg))
    // Check the container's accelerator keys:
        & ~ OLE-util-translate-accelerator(
              *server-object*, msg))
   // Message is not an accelerator key.
   TranslateMessage(msg); // Translate virtual keycodes
   DispatchMessage(msg);  // Dispatch message to window
  end if;
end while;
```

OLE-util-current-size *Function*

Signature **OLE-util-current-size** ***obj*** **=> (*width*, *height*)**

Arguments *obj* An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

Values *width* An instance of **<integer>**.

height An instance of **<integer>**.

Description Return the current size of the space allocated to the object in the container, as two integers representing the number of pixels.

These are the same values that would have been passed in the most recent call to **OLE-part-requested-size** or **OLE-part-change-size**.

OLE-util-HIMETRIC-size *Function*

Signature **OLE-util-HIMETRIC-size** ***obj*** **=> (*width*, *height*)**

Arguments *obj* An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

Values *width* An instance of **<integer>**.

height An instance of **<integer>**.

Description Returns the document window size — as from **OLE-util-current-size** — converted to **HIMETRIC** coordinates.

This function is probably only useful in a method for **OLE-part-get-data**, page 271.

OLE-util-close-server *Function*

Signature- **OLE-util-close-server *obj* => *ok***

Arguments *obj* An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

Values *ok* A status code, **$S-OK**.

Description Disconnects the server object (*obj*) from the container application, if it is connected to one. Does nothing otherwise. The return value is always **$S-OK**.

The return value is not usually useful.

OLE-util-in-place-active? *Function*

Signature **OLE-util-in-place-active? *obj* => *active?***

Arguments *obj* An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.

Values *active?* An instance of **`<boolean>`**.

Description Returns true if the server is currently activated in place, that is, currently activated with its image inside a container application window.

OLE-util-view-changed *Function*

Signature **`OLE-util-view-changed`** ***obj*** **`=> ()`**

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values None.

Description Tells the container that the server's image data has changed.

This will ensure that **`OLE-part-get-data`**, page 271, is invoked to copy the image to the container.

revoke-registration *Function*

Signature **`revoke-registration`** ***factory*** **`=> ()`**

Arguments *factory* An instance of **`<class-factory>`**.

Values None.

Description Before your server application terminates, make it call this function on the class factory instance to cancel the registration of the server object instance. In other words, this makes the server object no longer available for clients to connect to it. (The class is still available, but if a client tries to a new copy of the program will be initiated instead of using this one.)

This function does nothing if the argument is **#f** or **$null-instance**, or if the class factory was not registered anyway.

<class-factory> *Open concrete primary class*

Summary
This class implements the COM **IClassFactory** interface. Making an instance of it causes it to be registered with the system automatically, for use by potential clients.

Superclasses
<LPUNKNOWN>

Init-keywords

clsid: The COM Class ID that identifies the object the server provides. Required. This ID can be represented either as a string of 32 hexadecimal digits in the following format

"{*xxxxxxxx*-*xxxx*-*xxxx*-*xxxx*-*xxxxxxxxxxxx*}"

That is, where each *x* is a hexadecimal digit. For example:

"{e90f09e0-43db-11d0-8a04-02070119f639}"

Alternatively, the ID can be a **<REFCLSID>**, as returned from **make-GUID**.

You can get the ID value by generating a GUID with Harlequin Dylan's **create-id** utility. See "Creating GUID numbers" on page 106.

class: The Dylan class (usually a user-defined subclass of **<ole-server-framework>**) that is to be instantiated when the client (controller) requests it. Required.

args: Optional sequence of initialization arguments to be passed to **make** when instantiating the object. The default is to pass the same arguments as for the **<class-factory>** (**<ole-server-framework>** accepts and ignores those that are only for the factory, and **<class-factory>** ignores any that it does not recognize.) Note that **<ole-server-framework>** requires **typeinfo:** to be supplied.

server-module-handle:

Contains the **<hModule>** instance of the server DLL when invoked from an in-process server. This argument is not normally specified by the user, but is passed in automatically when the class factory is created from the in-process DLL entry-point. Note that if you do not specify an explicit **args** argument, then this argument will be passed into your object along with all the other initialization arguments.

server-context:

Indicates where the server is running. See the description of **create-dispatch** below for a list of possible values. Defaults to **$CLSCTX-LOCAL-SERVER**. You can use **$CLSCTX-INPROC-SERVER** instead to suppress external registration of the factory.

connection-flags:

Optional connection flags, controlling whether more than one client is allowed to invoke the same class factory (and hence use the same server process). The value is one of the following OLE constants:

$REGCLS-SINGLEUSE — Only one connection allowed. This is the default value.

$REGCLS-MULTIPLEUSE — Multiple connections allowed.

$REGCLS-MULTI-SEPARATE — Multiple connections allowed, separate control.

For further explanation of these constants, see the Microsoft OLE/COM API documentation for the function **CoClassRegisterClassObject**.

Description

This class implements the COM **IClassFactory** interface. Making an instance of it causes it to be registered with the system automatically, for use by potential clients.

A server application does not need to use the instance directly, except that it must call **revoke-registration** on the instance before terminating.

Any keyword arguments other than those documented above are passed in the **make** call when the Dylan class is instantiated. Note that **<ole-server-framework>** requires you to supply **typeinfo:**.

You can subclass **<class-factory>** (using **define COM-interface**) if desired for adding functionality (such as overriding the **IClassFactory/LockServer** and **terminate** methods).

6.7.7 OLE API hooks

The following group of functions provide hooks into the low-level OLE API to allow for possible extensions.

in-place-frame-info *Function*

Signature **`in-place-frame-info`** ***obj*** **`=>`** ***pInfo***

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values *pInfo* An instance of **`<LPOLEINPLACEFRAMEINFO>`**.

Description Returns the container application's frame information structure pointer.

container-IOleInPlaceFrame *Function*

Signature **`container-IOleInPlaceFrame`** ***obj*** **`=>`** ***pFrame***

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values *pFrame* An instance of **`<LPOLEINPLACEFRAME>`**.

Description Returns the container's **`IOleInPlaceFrame`** interface pointer.

The pointer is null when the server is not in-place active.

Note that this function does not call **`AddRef`** on the returned value.

server-IOleObject *Function*

Signature **`server-IOleObject`** ***obj* => *value***

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values *value* An instance of **`<IPOleObject>`**.

Description Returns the server object's **`IOleObject`** interface pointer, an instance of **`<IOleObject>`**.

Note that this function does not call **`AddRef`** on the returned value.

container-parent-window *Function*

Signature **`container-parent-window`** ***obj* => *window***

Arguments *obj* An instance of your subclass of **`<ole-server-framework>`** or **`<ole-in-process-server>`**.

Values *window* An instance of **`<HWND>`**.

Description Returns the container window that is the parent of the embedded part during in-place activation.

The value is an instance of the class **`<HWND>`** from the Win32 API libraries. For details, see the manual *C FFI and Win32 Reference* for details.

hatch-window *Function*

Signature **`hatch-window`** ***obj* => *window***

Arguments	*obj*	An instance of your subclass of **<ole-server-framework>** or **<ole-in-process-server>**.
Values	*window*	An instance of **<HWND>**.

Description

Returns the hatch border window provided by the server support library for in-place activation.

The value is an instance of the class **<HWND>** from the Win32 API libraries. For details, see the manual *C FFI and Win32 Reference* for details.

6.7.8 Initializing and registering in-process compound document servers

initialize-ole-server *Macro*

Summary

Required macro call for in-process server initialization and registration.

Signature

initialize-ole-server ***class class-ID prog-ID title-string*** **#rest** ***options***

Arguments	*class*	An instance of **<class>**.
	class-ID	An instance of **<REFGUID>** or **<string>**.
	prog-ID	An instance of **<string>**.
	title-string	An instance of **<string>**.
	options	Optional keyword and value pairs for additional registration options. See Description.

Values

None.

Description

In-process servers must call this macro at top level in order to set up some static initializations necessary for the server DLL's initialization and registration. (It is not an executable expression.)

The macro-expansion provides definitions for the OLE / COM functions **DllRegisterServer**, **DllUnregisterServer**, **DllGetClassObject**, and **DllCanUnloadNow**.

If your in-process server does not make this macro call, container applications will not be able to connect it.

You cannot use this macro more than once in a DLL library.

The *class* argument is your server's s subclass of **<ole-in-process-server>**, to be instantiated when the server is invoked.

The *class-ID* argument is the COM Class ID of the server object, an instance of **<REFGUID>** as returned from **make-GUID**, page 313, or string representation of the ID.

The *prog-ID* argument is the "programmatic identifier" string, as described for **register-ole-server**, page 254.

The *title-string* argument is the string that should appear in the container application's Insert Object dialog to identify this server application.

The *options* arguments are optional keyword and value pairs for additional registration options. The values accepted are any that are specified for **register-ole-server**, page 254.

6.7.9 Persistent storage for in-process compound document servers

All compound document servers, whether local or in-process, must define the methods for persistent storage described in Section 6.7.3 on page 248. Those methods are required for a server to implement the COM interface **IPersistStorage**.

In-process servers have the additional option of defining the following set of methods to implement the COM interface **IPersistStreamInit**, which is useful in the case where the container application calls **IPersistStreamInit** instead of **IPersistStorage**.

The following methods also serve as an implementation of the obsolete COM interface **IPersistStream**. **IPersistStream** differs from **IPersistStreamInit** only in that **IPersistStreamInit** has an extra COM method, **InitNew**, represented here by the Dylan method **OLE-part-init-new**.

OLE-part-max-save-size *Open generic function*

Summary
: Returns the maximum size, as an integer number, of bytes that are needed for the stream to which the persistent data will be written.

Signature
: **OLE-part-max-save-size** ***obj*** **=>** ***maximum-size***

Arguments
: *obj* An instance of your subclass of **<ole-in-process-server>**.

Values
: *maximum-size* An instance of **<integer>**.

Description
: Returns the maximum size, as an integer number, of bytes that are needed for the stream to which the persistent data will be written, or returns **#f** if it is not possible to make such an estimate. The default method always returns **#f**.

OLE-part-Save-To-Stream *Open generic function*

Summary
: Write the object's persistent data to the given stream.

Signature
: **OLE-part-Save-To-Stream** ***obj stream*** **=> ()**

Arguments	*obj*	An instance of your subclass of **`<ole-in-process-server>`**.
	stream	An instance of **`<storage-istream>`**.

Values None.

Description Write the object's persistent data to the given stream. This is similar to **`OLE-part-Save-To-Storage`**, page 251, except that the stream is provided instead of you having to create and remember it.

Note: There is no default method for this generic function. You must provide a method yourself.

OLE-part-Load-From-Stream *Open generic function*

Summary Read the object's persistent data from the given stream.

Signature **`OLE-part-Load-From-Stream`** ***obj stream*** **`=> ()`**

Arguments	*obj*	An instance of your subclass of **`<ole-in-process-server>`**.
	stream	An instance of **`<storage-istream>`**.

Values None.

Description Read the object's persistent data from the given stream. This is similar to **`OLE-part-Load-From-Storage`**, page 252, except that the stream is provided instead of you having to create and remember it.

Note: There is no default method for this generic function. You must provide a method yourself.

OLE-part-init-new

Open generic function

Signature **OLE-part-init-new** ***obj*** **=> ()**

Arguments *obj* An instance of your subclass of **<ole-in-process-server>**.

Values None.

Description The library calls either **OLE-part-init-new** or **OLE-part-Load-From-Stream** after the **initialize** method for the object. The default method does nothing. (However, the method is not called if the container is using the obsolete **IPersistStream** interface instead of **IPersistStreamInit**.)

OLE-part-dirty?-setter

Open generic function

Signature **OLE-part-dirty?-setter** ***dirty?*** ***obj*** **=>** ***dirty?***

Arguments *dirty?* An instance of **<boolean>**.

obj An instance of your subclass of **<ole-in-process-server>**.

Values *dirty?* An instance of **<boolean>**.

Description In addition to the method on **OLE-part-dirty?** that local servers must define, in-process server applications must supply a method on **OLE-part-dirty?-setter**. The application's dirty status should *not* be reset by **OLE-part-Save-To-Stream** or **OLE-part-Load-From-Stream**; instead, the library will call this setter function separately when appropriate.

Note: There is no default method for this generic function. You must provide a method yourself.

6.7.10 Other required methods for in-process servers

Other generic functions for which in-process compound server applications should provide methods follow.

OLE-part-draw

Open generic function

Signature `OLE-part-draw` ***obj hDC rect*** `=>` ***status***

Arguments

obj — An instance of your subclass of **`<ole-in-process-server>`**.

hDC — An instance of **`<HDC>`**.

rect — An instance of **`<LPRECTL>`**.

Values

status — An instance of **`<HRESULT>`**.

Description

In-process servers must implement this method to draw all of the document window display to the given device context, *hDC* (an instance of **`<HDC>`**), scaled and translated to fit within the given rectangle, *rect* (an instance of **`<LPRECTL>`**).

The image drawn by this is what is displayed by the container when the server is no longer active. Normally, this method should return **`$S-OK`** but it is possible to return an error status code that will be reported back to the container program.

7

The OLE-Control-Framework Library

7.1 Introduction

This document describes the facilities of the OLE-Control-Framework library.

You should use this library if you want to write an OLE control but you do *not* want to use the DUIM library to define its graphical user interface.

Otherwise, the DUIM-OLE-Control library offers a simpler means of implementing controls. See Chapter 3, “Compound Documents and OLE Controls in DUIM” for more information.

The OLE-Control-Framework library provides a broader range of facilities than the DUIM-OLE-Control library, so even if you are using DUIM-OLE-Control library you may find a need to use OLE-Control-Framework too.

7.2 OLE Controls

Controls — variously called OLE Controls, ActiveX Controls, and OCXs — are applications that combine the features of an in-process OLE compound document server and an OLE Automation server. Thus much of the process of implementing a control has been described in Chapter 4, “OLE Automation” and Chapter 6, “The OLE-Server Library”.

To implement an OLE control, use the OLE-Control-Framework library. The implementation process differs from using the OLE-Server library as follows:

- You must build your OLE control application as a Dynamic Link Library, as you would for an in-process OLE server application. However, following convention, you can use the file extension .OCX instead of .DLL.
- You must define a subclass of **`<OLE-control-framework>`**, page 301, instead of **`<ole-server-framework>`** or **`<ole-in-process-server>`**. (The class **`<OLE-control-framework>`** is a subclass of both **`<ole-in-process-server>`** and **`<simple-component-object>`**.)
- Define coclass type information, as for an OLE Automation server.
- Out-of-place activation will not be used.
- Since there is no main program, self-registration and creation of a class factory are handled by code generated by the macro **`initialize-ole-control`**, page 302.

Overall, the code for an OLE control will look something like this:

```
define COM-interface <my-object-class> (<ole-control-framework>)
 …
end;

define constant my-type-info =
  make(<coclass-type-info>, class: <my-object-class>, …);

…

initialize-ole-control(my-type-info, "my.prog.id");
```

7.2.1 Mnemonic keys

Besides the “accelerator” keys that apply when a server is active (see **`OLE-part-accelerators`**, page 269), OLE controls can also have “mnemonic” keys that will be processed by the control even if it is not already active. Mnemonics are implemented by using the following functions:

- **`OLE-part-mnemonics`**, page 303.
- **`OLE-part-on-mnemonic`**, page 303.
- **`OLE-util-key-change`**, page 304.

7.2.2 Ambient properties

OLE control containers typically make property information available to the controls they contain. Ambient properties allow a control to know about and adapt itself to the environment in which it is running.

Some properties are used directly by the OLE-Control-Framework library and you do not need to be concerned about them. (This includes **$DISPID-AMBIENT-SHOWHATCHING**.) Others are cached by OLE-Control-Framework so that you can access them easily by calling the following functions:

- **freeze-events?**, page 305
- **freeze-UI?**, page 306
- **OLE-util-locale**, page 306

The current value of any ambient property can be obtained like this:

```
let dispatch-interface = object.container-IDispatch;

unless ( null-pointer?(dispatch-interface) )
  let value = get-property(dispatch-interface, disp-id,
                           default: $unfound);
  unless ( value == $unfound )
    … // use the value
  end unless;
end unless;
```

The function **get-property** is from the OLE-Automation library; see page 193. A **default:** value must be supplied because containers are not required to support all properties. The particular property is identified by a Dispatch ID, which is one of the following constants:

$DISPID-AMBIENT-AUTOCLIP

True if container automatically clips the control display area.

$DISPID-AMBIENT-APPEARANCE

Control appearance:

0 => flat

1 => 3D

$DISPID-AMBIENT-BACKCOLOR

Background color. Convert the value with **OLE-util-translate-color**, page 309.

$DISPID-AMBIENT-FORECOLOR

Foreground color. Convert the value with **OLE-util-translate-color**, page 309.

$DISPID-AMBIENT-FONT

Font, as an **IDispatch** interface.

$DISPID-AMBIENT-DISPLAYNAME

Name of control for use in error messages.

$DISPID-AMBIENT-LOCALEID

Locale.

$DISPID-AMBIENT-MESSAGEREFLECT

Boolean value indicating whether the container wants to receive Windows messages such as **$WM-CTLCOLOR**, **$WM-DRAWITEM**, or **$WM-PARENTNOTIFY**.

$DISPID-AMBIENT-SCALEUNITS

A string (actually an instance of **<ole-array>**, page 173) naming the coordinate unit used by the container.

$DISPID-AMBIENT-TEXTALIGN

Text alignment:

0 => numbers to the right and text to the left

1 => left

2 => center

3 => right

4 => fill justify

$DISPID-AMBIENT-USERMODE

True if user mode, false if design mode.

$DISPID-AMBIENT-UIDEAD

True if user input should be ignored.

$DISPID-AMBIENT-SHOWGRABHANDLES

True to enable resize handles.

$DISPID-AMBIENT-DISPLAYASDEFAULT

True if control is the default button.

$DISPID-AMBIENT-SUPPORTSMNEMONICS

True if container supports mnemonics.

Alternatively, a control can ask to be notified whenever the value of an ambient property changes. This is done by defining a method on the function **OLE-part-ambient-properties**, page 308, to specify which properties are of interest, and defining one or more methods on **OLE-part-set-ambient-property**, page 308, to receive the values. The following functions are also related to ambient property support:

- **container-IDispatch**, page 307.
- **OLE-util-translate-color**, page 309.

7.3 The OLE-CONTROL-FRAMEWORK module

This section contains a reference entry for each item exported from the OLE-Control-Framework library's **OLE-Control-Framework** module.

OLE-Control-Framework also re-exports all the items from the OLE-Server library. See "The OLE-SERVER module" on page 243 for those reference entries.

<OLE-control-framework> *Open abstract class*

Summary The class of OLE controls.

Superclasses **<ole-in-process-server> <simple-component-object>**

Description The class of OLE controls.

initialize-ole-control *Macro*

Summary Required macro call for OLE control initialization and registration.

Macro call

```
initialize-ole-control (coclass-type-info prog-ID
                        #rest options)
```

Arguments

coclass-type-info An instance of **<coclass-type-info>**.

prog-ID An instance of **<string>**.

options Optional keyword and value pairs for additional registration options. See Description.

Description OLE control applications must call this macro at top level in order to set up some static initializations necessary for the control DLL / OCX's initialization and registration. (It is not an executable expression.)

The macro-expansion provides definitions for the OLE / COM functions **DllRegisterServer**, **DllUnregisterServer**, **DllGetClassObject**, and **DllCanUnloadNow**.

If your control does not make this macro call, control container applications will not be able to connect to it.

You cannot use this macro more than once in a DLL library.

The *coclass-type-info* argument is the type information describing the control's Automation functionality. It must be an instance of **<coclass-type-info>**, where the **class:** init-keyword passed to **make** specified your control's subclass of **<OLE-control-framework>**.

The *prog-ID* argument is the programmatic identifier string, as described for **register-ole-server**, page 254.

The *options* arguments are optional keyword and value pairs for additional registration options. The values accepted are any that are specified for **register-ole-server** or **register-automation-server**, page 206, except that **title:** is accepted in place of the *title-string* positional argument (it defaults from the **name:** of the type info) and the default value for the **verbs:** option is:

```
vector(vector($OLEIVERB-PRIMARY, "&Edit", $MF-ENABLED,
              $OLEVERBATTRIB-ONCONTAINERMENU))
```

OLE-part-mnemonics *Open generic function*

Summary Returns the handle of an accelerator table specifying the application's mnemonic keys.

Signature **OLE-part-mnemonics** ***obj*** **=>** ***table***

Arguments

obj An instance of your subclass of **<OLE-control-framework>**.

Values

table An instance of **false-or(<HACCEL>)**.

Description Your control can define a method on this function to return the handle of an accelerator table specifying the application's mnemonic keys (instance of **<HACCEL>**), or **#f** if the application does not use any mnemonics. The default method returns **#f**.

Control containers use this function to forward keyboard events to a control that is not currently active.

The control indicates which keys it is prepared to handle, but the container is not required to honor the request. Various containers may have different policies about what kind of keystrokes they pass through.

OLE-part-on-mnemonic *Open generic function*

Summary The control container calls this function when the user presses one of the designated mnemonic keys.

Signature **OLE-part-on-mnemonic** ***obj vkey pMsg*** **=> ()**

Arguments

obj An instance of your subclass of **<OLE-control-framework>**.

vkey An instance of **<integer>**.

pMsg An instance of **<LPMSG>**.

Values None.

Description This function is called by the container when one of the designated mnemonic keys is pressed. The *vkey* argument is the virtual key code. The *pMsg* argument is a pointer to the actual keypress message, in case additional information might be needed from it.

The default method sends the corresponding **$WM-SYSCOMMAND** message by doing:

```
TranslateAccelerator(OLE-part-command-window(obj),
                     OLE-part-mnemonics(obj),
                     pMsg);
```

The application can define an override method if that is not appropriate.

OLE-util-key-change *Function*

Summary The application should call this function when it wants to change the set of mnemonics supported.

Signature **OLE-util-key-change** ***obj*** **=> ()**

Arguments

obj An instance of your subclass of **<OLE-control-framework>**.

Values None.

Description The application should call this function when it wants to change the set of mnemonics supported. This will force , page 303, to be called again to get the new information.

OLE-util-on-focus *Function*

Summary The control should call this function to report to the container that it has received or lost the focus, if these events might change the way the container would handle the control.

Signature `OLE-util-on-focus` ***obj got-focus?*** `=> ()`

Arguments

obj An instance of your subclass of `<OLE-control-framework>`.

got-focus? An instance of `<boolean>`.

Values None.

Description The control should call this function to report to the container that it has received or lost the focus, if these events might change the way the container would handle the control.

For example, a button might need to be displayed differently when it becomes the default button, or the handling of the Return and Escape keys may depend on which control has the focus.

If the control has received the focus, it should pass `#t` as the *got-focus?* argument. If it has lost the focus, it should pass #f to *got-focus?*

freeze-events? *Function*

Summary Returns true when the container is not accepting events from the OLE control.

Signature **`freeze-events?`** ***obj*** **`=>`** ***frozen?***

Arguments *obj* An instance of your subclass of **`<OLE-control-framework>`**.

Values *frozen?* An instance of **`<boolean>`**.

Description Returns true when the container is not accepting events (that is, **`IDispatch`** dispatch method calls) from the OLE control.

freeze-UI? *Function*

Summary When this function returns true, the control should ignore user input events.

Signature **`freeze-UI?`** ***obj*** **`=>`** ***frozen?***

Arguments *obj* An instance of **`<OLE-control-framework>`**.

Values *frozen?* An instance of **`<boolean>`**.

Description When this function returns true, the control should ignore user input events such as mouse clicks. (This corresponds to the ambient property **`$DISPID-AMBIENT-UIDEAD`**.)

OLE-util-locale *Function*

Summary Returns the Windows locale code obtained from the container.

Signature **`OLE-util-locale`** ***obj*** **`=>`** ***lcID***

Arguments *obj* An instance of your subclass of **`<OLE-control-framework>`**.

Values *lcID* An instance of **<integer>**.

Description Returns the Windows locale code obtained from the container. This information could be used to adapt messages to the user's language.

Refer to the documentation for the OLE-Automation library for more information about locale codes. See "Internationalization" on page 171.

If the container does not supply a value, it defaults to **$LOCALE-USER-DEFAULT**.

container-IDispatch *Function*

Summary Returns the container's **IDispatch** interface pointer.

Signature **container-IDispatch *obj* => *idisp***

Arguments *obj* An instance of your subclass of **<ole-server-framework>**.

Values *idisp* An instance of **<LPDISPATCH>**, page 172.

Description Returns the container's **IDispatch** interface pointer. (This is just a slot accessor -- it does not call **AddRef** on the returned value.) If not null, this can be used as the first argument to **get-property** (from the OLE-Automation library, see page 193) to obtain the current values of ambient properties from the container. The pointer may be null if the container does not support ambient properties or if this is called before the control is fully initialized.

OLE-part-ambient-properties *Open generic function*

Summary Returns a sequence of Dispatch IDs of the ambient properties that the control application would like to be notified of changes to.

Signature **OLE-part-ambient-properties** ***obj*** **=>** ***properties***

Arguments *obj* An instance of your subclass of **<OLE-control-framework>**.

Values *properties* An instance of **<sequence>**.

Description Returns a sequence Dispatch IDs of the ambient properties that the control application would like to be notified of changes to. This notification consists of the container calling **OLE-part-set-ambient-property** upon a change to any of the ambient properties in the sequence.

The default method returns an empty sequence. If you would like to be notified of changes to particular ambient properties, define a method on this function to return a sequence containing the Dispatch IDs for those properties.

If you are defining a method on a subclass of **<DUIM-OCX>**, page 146 (that is, when using the DUIM-OLE-Control library), use **union** to merge your values with those returned by **next-method**.

OLE-part-set-ambient-property *Open generic function*

Summary The container calls this function to report the value of an ambient property, either at start-up or whenever the value changes.

Signature **OLE-part-set-ambient-property** ***obj disp-ID value*** **=> ()**

Arguments	*obj*	An instance of **`<OLE-control-framework>`**.
	disp-ID	An instance of **`<integer>`**.
	value	An instance of **`<object>`**.

Values None.

Description This function is called to report the value of an ambient property, either at start-up or whenever the value changes. This will be called only for properties that are supported by the container and have either been requested by being included in the value returned from **`OLE-part-ambient-properties`** or are used directly by the library.

Your control application should provide methods as needed for the properties it wishes to receive. You should call **`next-method`** in case the library also wants to handle the property.

Note that you should use **`OLE-util-translate-color`** if the value is to be interpreted as a color.

OLE-util-translate-color *Function*

Summary Translates an ambient property color value to a Win32 color code.

Signature **`OLE-util-translate-color`** ***obj ambient-color*** **`=>`** ***color-ref***

Arguments	*obj*	An instance of your subclass of **`<OLE-control-framework>`**.
	ambient-color	An instance of **`<object>`**.
Values	*color-ref*	An instance of **`<integer>`**.

Description This should be used for converting the value of the ambient properties for foreground or background color to a Win32 color value. For example:

```
define method OLE-part-set-ambient-property
  (obj :: <my-object-class>,
   disp-id == $DISPID-AMBIENT-FORECOLOR,
   value :: <object>) => ()
  let color = OLE-util-translate-color(obj, value);

  …
```

8

Basic COM Integration

8.1 Introduction

This chapter provides reference documentation for various functions, methods, macros, and classes that are fundamental to or useful in building all kinds of OLE/COM application with Harlequin Dylan. It is organized by topic.

8.2 Basic COM interface and data representation in Dylan

<LPUNKNOWN> *Open abstract instantiable primary class*

Summary The abstract superclass of all OLE/COM interface objects.

Superclasses **`<object>`**

Description The abstract superclass of all OLE/COM interface objects.

8.3 GUID utilities

<REFGUID> *Open class*

Summary The class of globally unique identifiers (GUIDs) in COM.

Superclasses `<C-pointer>`

Description The class of globally unique identifiers (GUIDs) in COM.

Instances of this class provide the Dylan representation of C values that the Microsoft OLE / COM libraries use to represent GUIDs.

Note that GUIDs are, abstractly, just strings of numbers; it is only because they are represented specially on the C side that we have this class. GUIDs are also known as universally unique identifiers or UUIDs.

The function **`make-GUID`** returns an instance of this class, and many functions concerning type information, or server and control registration, expect or can accept an instance of this class as an argument.

<REFCLSID> *Open class*

Summary An alias for **`<REFGUID>`**, the class of globally unique identifiers.

Superclasses `<C-pointer>`

Description An alias for **`<REFGUID>`**, the class of globally unique identifiers (GUIDs) in COM. It is a convenient alias for the use of GUIDs as Class IDs.

make-GUID *Function*

Summary Creates an object representing a globally unique identifier (GUID).

Signature **`make-GUID`** ***l w1 w2 b1 b2 b3 b4 b5 b6 b7 b8*** **`=>`** ***ID***

Arguments

l	An instance of **`<machine-word>`** or **`<double-integer>`**. (32 bits.)
w1	An instance of **`<integer>`**. (16 bits.)
w2	An instance of **`<integer>`**. (16 bits.)
b1	An instance of **`<integer>`**. (8 bits.)
b2	An instance of **`<integer>`**. (8 bits.)
b3	An instance of **`<integer>`**. (8 bits.)
b4	An instance of **`<integer>`**. (8 bits.)
b5	An instance of **`<integer>`**. (8 bits.)
b6	An instance of **`<integer>`**. (8 bits.)
b7	An instance of **`<integer>`**. (8 bits.)
b8	An instance of **`<integer>`**. (8 bits.)

Values

ID	An instance of **`<REFGUID>`**.

Description Creates an object representing a globally unique identifier (GUID) from the arguments.

This function creates and initializes a C structure and returns a Dylan object representing it. The structure is in the form that the Microsoft COM libraries expect to see when passing GUIDs around as arguments or return values.

This function does not create the GUID value itself. You create the GUID's value with a separate utility, **`create-id`**. (See below.)

All argument values are hexadecimal literals. The first argument is a 32-bit value, the second and third are 16-bit values, and the remaining eight are 8-bit (single byte) values.

You will need this function whenever you need to represent GUID values. For instance, OLE server applications and controls must be registered in the Windows Registry with a unique (and fixed) COM Class ID, so that clients can invoke them. Class IDs are GUIDs. To give your application its Class ID, you must generate a GUID for it. Other items, such as Automation dispinterfaces, also require GUIDs.

This function returns the value *ID*, which is an instance of the class **`<REFGUID>`** (which has the alias **`<REFCLSID>`**).

The **`make-GUID`** function creates the C structure necessary to represent a GUID. However, it does *not* create the unique hexadecimal values that constitute the GUID. You must do that with a once-only invocation of a special utility program provided with Harlequin Dylan called **`create-id`**. You can find the **`create-id`** utility in your Harlequin Dylan installation's Bin folder.

To create a GUID value, run the **`create-id`** utility at an MS-DOS prompt. The utility takes a single integer argument, which is the number of GUIDs to generate. (The default is one GUID.)

The utility writes the GUIDs it generates to the standard output in the form of string literals that can be pasted into Dylan source code. For example, given the following value from **`create-id`**:

```
MS-DOS> create-id
"{113f2c00-f87b-11cf-89fd-02070119f639}"
```

you need to split the value into parts as follows, adding the prefix #x to each. Then you call **`make-GUID`** on those values:

```
define constant $my-class-ID =
  make-GUID(#x113f2c00, #xf87b, #x11cf, #x89, #xfd,
            #x02, #x07, #x01, #x19, #xf6, #x39);
```

See “GUIDs: Globally unique identifiers” on page 99 for more on GUIDs and when to use **make-GUID**.

as <string> *GUID* => *string* — *G.f. method*

Summary Converts the internal representation of a GUID (instance of **<REFGUID>**) to a representation as a string, by calling the Microsoft library function **StringFromGUID2**.

Signature **as <string>** ***GUID*** **=>** ***string***

Arguments

<string>	The class name **<string>**.
GUID	An instance of **<REFGUID>**.

Values

string	An instance of **<string>**.

Description Converts the internal representation of a GUID (instance of **<REFGUID>**) to a representation as a string, by calling the Microsoft library function **StringFromGUID2**. The string has the form:

"{*xxxxxxxx*-*xxxx*-*xxxx*-*xxxx*-*xxxxxxxxxxxx*}"

where each *x* is a hexadecimal digit. For example:

"{113f2c00-f87b-11cf-89fd-02070119f639}"

as <REFGUID> *string* => *GUID* — *G.f. method*

Summary Calls the Microsoft library function **CLSIDFromString** to convert *string* to a GUID.

Signature **as <REFGUID>** ***string*** **=>** ***GUID***

Arguments

<REFGUID>	The class name **<REFGUID>**.
string	An instance of **<string>**.

Values	*GUID*	An instance of **`<REFGUID>`**.

Description

Calls the Microsoft library function **`CLSIDFromString`** to convert *string* to a GUID. The string can represent the GUID in one of two forms. The first form is the standard hexadecimal string form, as follows:

"{*xxxxxxxx*-*xxxx*-*xxxx*-*xxxx*-*xxxxxxxxxxxx*}"

For example:

```
"{113f2c00-f87b-11cf-89fd-02070119f639}"
```

The enclosing braces are a required part of the syntax; letters may be either upper or lower case.

The second form is as a programmatic identifier or Prog ID — a registered, unique symbolic alias for a numeric GUID. The Prog ID is looked up in the Windows registry and the corresponding GUID is returned.

This function signals an error if the string is not in the expected format, and cannot be found in the registry either.

The value returned is a C structure pointer, so when you are finished with it you must call the function **`destroy`** on it to ensure that the memory it uses is recycled.

8.4 COM IUnknown interface methods

AddRef *Open generic function*

Summary

Implements the COM method **`AddRef`** from the COM interface **`IUnknown`**.

Signature

`AddRef` ***obj*** **`=>`** ***count***

Arguments	*obj*	An instance of **`<interface>`**.

Values *count* An instance of **`<integer>`**.

Description Implements the COM method **`AddRef`** from the COM interface **`IUnknown`**. Adds a reference and returns the new value of *obj*'s count.

Release *Open generic function*

Summary Releases the given COM interface. Implements the COM method **`Release`** from the COM interface **`IUnknown`**.

Signature **`Release`** ***obj*** **`=>`** ***count***

Arguments *obj* An instance of **`<interface>`**.

count An instance of **`<integer>`**.

Description Releases the COM interface *obj*. Implements the COM method **`Release`** from the COM interface **`IUnknown`**. Decrements the reference count and returns the new count.

For interfaces implemented in Dylan as subclasses of **`<IUnknown>`**, **`terminate`** is called when the reference count is decremented to 0.

QueryInterface *Function*

Summary Given one COM interface, find another interface supported by the server application that has a particular GUID.

Signature **`QueryInterface`** ***interface interface-ID*** **`=>`** ***status new-interface***

Arguments *interface* An instance of **`<LPUNKNOWN>`**.

interface-ID An instance of **`<REFGUID>`**.

Values *status* An instance of **`<SCODE>`**.

object	An instance of **`<LPUNKNOWN>`**.

Description

Given one COM interface (*interface*), find another interface supported by the server application that has the GUID *interface-ID*. This function implements the basic COM method **`QueryInterface`**, from the **`IUnknown`** interface.

If the server supports an interface with the given GUID, the first return value from this function (*status*) will satisfy the predicate **`SUCCEEDED?`**, and the second value (*new-interface*) will be the interface that was found.

The first value is an OLE status value, an instance of **`<SCODE>`**. If the server does not support an interface with the supplied GUID, the first return value will be **`$E-NOINTERFACE`**.

Note: You should call **`Release`** on *object* when you have finished using it.

8.5 Error handling

If an error occurs in an OLE protocol operation, the OLE library signals a condition of class **`<ole-error>`**. You can signal an **`<ole-error>`** yourself with the **`check-ole-status`** function.

<ole-error> *Open concrete primary class*

Summary

The class of OLE error codes.

Superclasses

`<error>`

Init-keywords

`status:`	An instance of **`<SCODE>`**. Required.
`context:`	An instance of **`false-or(<string>)`**. Default value: **`#f`**.
`instance:`	An instance of **`<interface>`**.

args: An instance of **<sequence>**. Default value: #**()**.

Description

The class of OLE error codes.

Rather than instantiate an error directly with **make**, you can also use the convenience function **check-ole-status**, or **ole-error** to create and signal an instance of this class.

You can use the following accessor functions on the error object:

ole-error-status

Returns the low-level status code.

ole-error-context

Returns the name of the function that returned the code.

ole-error-instance

Returns the interface that the function was acting upon.

ole-error-args

Returns a **<sequence>** of additional function arguments which the code that called **ole-error** or **check-ole-status** might have supplied in its call.

ole-error *Function*

Summary

Signals an **<ole-error>** with the specified parameters.

Signature

ole-error ***status context instance*** **#rest** ***args***

Arguments

status An instance of **<SCODE>**. Required.

context	An instance of **`false-or(<string>)`**. Default value: **`#f`**.
instance	An instance of **`<interface>`**.
args	An instance of **`<sequence>`**. Default value: **`#()`**.

Values None; the function does not return.

Description Signals an **`<ole-error>`** with the specified parameters.

The *context* argument is intended to be the name of the function that returned the status code (*status*, an instance of **`<SCODE>`**); the *instance* argument is the interface the function was operating on; and you can list relevant function arguments for information purposes by passing them as *args*.

check-ole-status *Function*

Summary Calls **`ole-error`** to signal an error condition if a status value indicates an error has occurred.

Signature **`check-ole-status`** ***status context instance*** **`#rest`** ***args*** **`=> ()`**

Arguments

status	An instance of **`<SCODE>`**. Required.
context	An instance of **`false-or(<string>)`**. Default value: **`#f`**.
instance	An instance of **`<interface>`**.
args	An instance of **`<sequence>`**. Default value: **`#()`**.

Values None.

Description Calls **`ole-error`** to signal an error condition, if *status* indicates an error. This function returns no values.

In some contexts, the support library handles such errors by reporting *status* back to the container application. This is done using the macro

returning-error-status ?*body* end

The **returning-error-status** macro returns **$S-OK** if all of the *body* forms were successfully executed, or the error status code if an **<ole-error>** was signalled during execution of *body*, a Dylan body$_{\mathrm{bnf}}$.

The arguments for this function as the same as those for **ole-error**.

ole-error-status *Function*

Signature **ole-error-status** ***ole-error*** **=>** ***status***

Arguments *ole-error* An instance of **<ole-error>**.

Values *status* An instance of **<HRESULT>** or **<SCODE>**.

Description Returns the OLE status code from an instance of **<ole-error>**.

This function is the accessor for **<ole-error>**’s **status:** slot.

ole-error-context *Function*

Signature **ole-error-context** ***ole-error*** **=>** ***function-name***

Arguments *ole-error* An instance of **<ole-error>**.

Values *function-name* An instance of **<string>**.

Description Returns a string identifying the function involved in the error, usually the function name.

The *ole-error* argument is an instance of **`<ole-error>`**.

This function is the accessor for **`<ole-error>`**'s **`context:`** slot.

ole-error-instance *Function*

Signature **`ole-error-instance`** ***ole-error*** **`=>`** ***ole-interface***

Arguments *ole-error* An instance of **`<ole-error>`**.

Values *ole-interface* An instance of **`false-or(<interface>).`**

Description Returns the OLE interface instance involved in the error, or **`#f`** if you created the error instance with that value.

This function is the accessor for **`<ole-error>`**'s **`instance:`** slot.

ole-error-args *Function*

Signature **`ole-error-args`** ***ole-error*** **`=>`** ***values***

Arguments *ole-error* An instance of **`<ole-error>`**.

Values *values* An instance of **`<sequence>`**.

Description Returns a sequence of any other values that were passed in the call to instantiate *ole-error*. There is no built-in use for these values.

This function is the accessor for **`<ole-error>`**'s **`args:`** slot.

9

OLE FFI Facilities

9.1 Introduction

This chapter discusses the low-level Harlequin Dylan libraries that provide an interface to the C-based OLE/COM libraries in the Microsoft Win32 API.

Harlequin built these Dylan libraries using the Harlequin Dylan C-FFI library, applying a consistent scheme for mapping Win32's C names to names fitting Dylan style and coding conventions.

Using these libraries, you can write Dylan applications that use OLE in almost exactly the same way as a C++ program would — assuming it used direct OLE/COM calls instead of the Microsoft Foundation Classes. Most of the function, type, variable, and constant names documented in the Microsoft OLE/COM specifications are defined in Dylan for you to use.

9.2 Dylan and Win32 library correspondences

The following table shows lists the Win32 libraries for which corresponding Dylan libraries are available.

Dylan library	C header	Link library	Runtime library
COM	OBJBASE.H	OLE32.LIB,UUID.LIB	OLE32.DLL
OLE	OLE2.H	OLE32.LIB,UUID.LIB	OLE32.DLL
OLE-Automation	OLEAUTO.H	OLEAUT32.LIB	OLEAUT32.DLL
OLE-Dialogs	OLEDLG.H	OLEDLG.LIB	OLEDLG.DLL
OLE-Control	OLECTL.H	OLEPRO32.LIB	OLEPRO32.DLL

Table 9.1 Harlequin Dylan libraries and corresponding Windows files.

9.3 C-to-Dylan name-mapping scheme

This section describes the scheme Harlequin used to map the C names in the Microsoft OLE/COM API to names fitting Dylan style and coding conventions.

- Class and type names are enclosed in angle brackets. For example, **HRESULT** becomes **<HRESULT>**. For Dylan classes representing C pointers, the Dylan **=** operator compares the pointer addresses, and **==** is not likely to be useful. (Comparison of structure contents, where applicable, is done by the same functions as in the C API.)
- Names of constants are prefixed by a dollar sign ($). For example, **NOERROR** becomes **$NOERROR**.
- Underscores are replaced by hyphens. For example, **MAKE_HRESULT** becomes **MAKE-HRESULT** and **S_OK** becomes **$S-OK**. (However, hyphens are not inserted between capitalized words.)
- Functions whose only effect is to return a true-or-false value have a question mark (?) appended to their names. For example: **SUCCEEDED?**, **FAILED?**, and **IsEqualIID?**.
- Where an interface function is documented under a name such as, for example, **IClassFactory::CreateInstance**, the corresponding Dylan generic function is called **IClassFactory/CreateInstance**. This means that function names generally need to be qualified by their interface

names in Dylan, although they are not in C or C++. This is necessary because there are several names that are used in different interfaces with incompatible argument lists, so that the same generic function cannot be used. However, for convenience, the following alias names are defined:

```
define constant QueryInterface = IUnknown/QueryInterface;

define constant AddRef = IUnknown/AddRef;

define constant Release = IUnknown/Release;
```

- Where a C / C++ function takes a pointer argument as a place to store a result value (when that value is a pointer or integer, not when filling in fields in a structure), the corresponding Dylan function uses multiple return values to return such output parameters following the original function return value. For example, where C++ code says:

```
status = obj->QueryInterface(IID_Ifoo, & result);
```

  the equivalent Dylan code is:

```
let ( status :: <SCODE>, result :: <Interface> ) =
  QueryInterface(obj,$IID-Ifoo);
```

- The multitude of integer data types in C code (**int**, **long**, **unsigned**, **ULONG**, **DWORD**, and so on) are all designated as **<integer>** (or some appropriate subrange thereof) in Dylan method argument types. However, in some cases, values that are too large to be represented as an **<integer>** are represented as a **<machine-word>** instead.

  The names **<SCODE>** and **<HRESULT>** are defined as Dylan data types because they are not used as integers even though they are implemented as such in C. (In Dylan, they are actually aliases for **<machine-word>**.)

- The C type **BOOL** is mapped to **<boolean>** in Dylan. Use **#t** and **#f** instead of **TRUE** and **FALSE**.

- Because slot names are not in a separate name space in Dylan, the names of C structure fields will have the suffix **-value** added to form the name of the Dylan accessor function. For example, the C statement:

```
pt->x = x;
```

  becomes in Dylan:

```
pt.x-value := x;
```

- The class **<Interface>** is an abstract class that includes all OLE interfaces, regardless of whether they are implemented in C or Dylan. Thus this is the appropriate declared type for a variable holding an arbitrary interface, such as returned by **QueryInterface** or **CreateInstance**. Classes such as **<LPUNKNOWN>**, **<LPOLECONTAINER>**, **<LPOLEOBJECT>**, and so on represent specific interfaces which could be implemented in either C or Dylan; conceptually these are subclasses of **<Interface>**, but they are currently actually implemented as aliases of **<Interface>** instead of as distinct types. . The C-FFI library's function **pointer-cast** can be used to convert an **<Interface>** to one of the more specific types. The classes **<IUnknown>**, **<IOleContainer>**, **<IOleObject>**, and so on are subclasses used for interfaces implemented in Dylan. Thus, the class hierarchy looks (conceptually) like:

```
   <Interface>
       |
    <LPUNKNOWN>
     /       \
    /       <IUnknown>
<LPOLEOBJECT>  /     \
   /   \      /       \
  /   <IOleObject>    ...
 ...       |
    <my-OleObject>
```

  where **<my-OleObject>** represents a user-defined Dylan class that provides an implementation of the **IOleObject** interface. Classes **<IUnknown>** and **<IOleObject>** do not have any direct instances.

- The function call **null-pointer(<LP…>)** is used to create a null pointer of a particular interface type, where C code would just use **NULL**. The constant **$NULL-interface** is provided as a null instance of type **<Interface>**, which is not otherwise directly instantiable. The function **null?** can be used to test whether an instance of **<Interface>** (or any other C pointer type) is null. It also returns **#t** if the argument is **#f**. It is not valid to use **null-pointer** on a Dylan implementation class (that is, any subclass of **<IUnknown>**).

9.4 How to use the Dylan libraries

As in C++, an OLE application is written by defining subclasses for the various interface classes to be supported, and the methods to implement the prescribed behavior. Multiple inheritance of interface classes is not allowed, but **`<IUnknown>`** is implicitly included as a superclass of all of the other interfaces. The library provides a complete implementation of the **`IUnknown`** interface, so the user is not required to define the methods **`QueryInterface`**, **`AddRef`**, and **`Release`**, although they may be overridden if necessary. When **`make`** is called to instantiate an interface class, the optional keyword **`controlling-unknown:`** may be used to designate the object handling the **`IUnknown`** protocol on behalf of an aggregate object.

Other methods for which default implementations are provided: **`LockServer`**

For example, where C++ might say:

```
interface myOLEobject : public IOleObject
  {
…
};
```

the corresponding Dylan code would be:

```
define COM-interface <myOLEobject> ( <IOleObject> )
…
end;
```

The macro **`define COM-interface`** has the same syntax and semantics as **`define class`**, but compiler limitations require it to be used to perform special handling for interface classes.

For convenience, it is permissible to call **`AddRef`** or **`Release`** on a null interface, so it is not necessary to check first.

`<IUnknown>` also implements a function called **`SubRef`**, which is like **`Release`** in that it undoes the effect of a call to **`AddRef`**, but differs from **`Release`** in that it will not terminate the object if the count is returned to 0. This is sometimes needed within an **`initialize`** method to return the reference count to 0 before the **`AddRef`** is performed by the caller of **`make`**. (In C / C++ code, this situation is typically handled by directly incrementing and decrementing the reference count slot, without using **`AddRef`** or **`Release`**, but the Dylan implementation does not directly expose that slot.)

If the user class is to define a new interface (as opposed to implementing a pre-defined interface), then for **QueryInterface** to recognize the new interface, it is sufficient to define an interface ID and a method like:

```
define method initialize ( self :: <IFoo> )
  next-method();
  add-interface(self, $IID-IFoo);
end initialize;
```

However, the methods of the new interface will not be callable from code in other languages without the use of additional tools that are not currently provided. Such tools are expected to be needed, but are not yet specified.

For defining the interface ID, where C code would do like:

```
DEFINE_GUID(GUID_SIMPLE, 0xBCF6D4A0, 0xBE8C, 0x1068, 0xB6, 0xD4,
              0x00, 0xDD, 0x01, 0x0C, 0x05, 0x09);
```

in Dylan do:

```
define constant $GUID-SIMPLE :: <REFGUID> =
  make-GUID(#xBCF6D4A0, #xBE8C, #x1068, #xB6, #xD4, #x00, #xDD,
            #x01, #x0C, #x05, #x09);
```

Alternatively, the internal **GUID** can also be converted from the string representation like this:

```
define constant $GUID-SIMPLE :: <REFGUID> =
  as(<REFGUID>, "{BCF6D4A0-BE8C-1068-B6D4-00DD010C0509}");
```

There is also an **as** method for converting a GUID to a **<string>**.

Since Dylan does not have destructors, any cleanup code that would have been done in a C++ destructor must instead be done by providing a method on generic function **terminate**. The default method for **Release** will call **terminate** when the object's reference count is decremented to zero. It may also be called before the object is garbage collected, if GC finalizations are available and enabled.

For example, where a C++ OLE application class might do:

```
myApp::~myApp()
  {
    if (IsInitialized())
      OleUninitialize();
    DestroyWindow(m_hAppWnd);
}
```

the corresponding Dylan code would look like:

```
define method terminate (this :: <my-app>) => ();
  next-method();
  if ( IsInitialized() )
    OleUninitialize();
  end if;
  DestroyWindow(this.m_hAppWnd);
  values();
end terminate;
```

The Dylan function **`pointer-value`** can be used to convert between a Dylan integer and a **`LARGE_INTEGER`** or **`ULARGE_INTEGER`**. For example:

```
let li :: make( <PLARGE-INTEGER> );
  pointer-value(li) := 0;
```

allocates a **`LARGE_INTEGER`** and sets its value to 0, without needing to be concerned with the individual fields of the internal representation.

Note that this makes the C macros **`LISet32`** and **`ULISet32`** unnecessary in Dylan.

When an interface pointer is received from an API call, it could represent an interface implemented in either C/C++ or Dylan, so any calls to its methods need to first go through the C/C++ method table, and then through the Dylan generic function dispatch if it is actually implemented in Dylan. If the interface is going to be used in a series of method calls, it may be more efficient to first call function **`dylan-interface`** on it to obtain a direct Dylan representation of the interface that uses Dylan dispatch directly. The argument is returned unchanged if it is not a Dylan interface. For example:

```
let interface = dylan-interface( object.foo-interface );
  foo(interface, ...);
  bar(interface, ...);
```

Some other helper functions provided to make it easier to use Dylan data types:

IStream/Write-integer(stream, integer-value) => (status, count)

Calls IStream / Write to output the integer-value in a 4-byte field.

IStream/Read-integer(stream) => (status, integer-value, count)

Calls IStream / Read to read a value written by IStream / Write-integer.

For convenience, the class **`<LPSTGMEDIUM>`** has **`pointer-value`** and **`pointer-value-setter`** methods provided which use Dylan's run-time typing to automatically manage the **`tymed`** field.

Index

E

F

G

K

L