(51) International Patent Classification 5: G06F 9/44
(11) International Publication Number: WO 94/09428
(43) International Publication Date: 28 April 1994 (28.04.94)

(21) International Application Number: PCT/US93/09673
(22) International Filing Date: 8 October 1993 (08.10.93)
(30) Priority data:
07/959,056 9 October 1992 (09.10.92) US

(71) Applicant: MICROSOFT CORPORATION [US/US];
One Microsoft Way, Redmond, WA 98052-6399 (US).

(72) Inventors: ACTON, Colin, J.; 8623 N.E. 127th Street, Kirkland, WA 98034 (US).
CARON, Ilan, G.; 8411 142nd Avenue N.E., Redmond, WA 98052 (US).
CARVER, Alan, W.; 15940 S.E. 1st Street, Bellevue, WA 98008 (US).
CORNET, Tom; 2085 Oakmont Way, Eugene, OR 97401 (US).
WOOLF, Michael, J.; 12126 215th Place S.E., Snohomish, WA 98290 (US).

(74) Agents: PIRIO, Maurice, J. et al.; Seed and Berry, 6300 Columbia Center, 701 Fifth Avenue, Seattle, WA 98104-7092 (US).


Published
With international search report.

(54) Title: METHOD AND SYSTEM FOR INTERFACING TO A TYPE LIBRARY

(57) Abstract

A method and system for interfacing to a type library (204) are provided. In a preferred embodiment, the present invention defines an interface (203) to a type library. The interface has a plurality of interface methods (404-411) through which type information can be stored in and retrieved from the type library. A plurality of implementations of the defined interfaces are provided. A computer program is compiled using the defined interfaces to access the type information. When the compiled computer program is executed, it accesses the type information using one of the plurality of implementations. In a preferred embodiment, a type library contains type information to allow a compiler to bind to an instance of type at compile time (early binding).
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>Austria</td>
<td>FR</td>
<td>France</td>
<td>MR</td>
<td>Mauritania</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AU</td>
<td>Australia</td>
<td>GA</td>
<td>Gabon</td>
<td>MW</td>
<td>Malawi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BB</td>
<td>Barbados</td>
<td>GB</td>
<td>United Kingdom</td>
<td>NE</td>
<td>Niger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BE</td>
<td>Belgium</td>
<td>GN</td>
<td>Guinea</td>
<td>NL</td>
<td>Netherlands</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td>Burkina Faso</td>
<td>GR</td>
<td>Greece</td>
<td>NO</td>
<td>Norway</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG</td>
<td>Bulgaria</td>
<td>HU</td>
<td>Hungary</td>
<td>NZ</td>
<td>New Zealand</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BJ</td>
<td>Benin</td>
<td>IE</td>
<td>Ireland</td>
<td>PL</td>
<td>Poland</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>Brazil</td>
<td>IT</td>
<td>Italy</td>
<td>PT</td>
<td>Portugal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BY</td>
<td>Belarus</td>
<td>JP</td>
<td>Japan</td>
<td>RO</td>
<td>Romania</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>Canada</td>
<td>KP</td>
<td>Democratic People's Republic of Korea</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CF</td>
<td>Central African Republic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td>Congo</td>
<td>KR</td>
<td>Republic of Korea</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>Switzerland</td>
<td>KZ</td>
<td>Kazakhstan</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CI</td>
<td>Côte d'Ivoire</td>
<td>LI</td>
<td>Liechtenstein</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM</td>
<td>Cameroon</td>
<td>LK</td>
<td>Sri Lanka</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CN</td>
<td>China</td>
<td>LU</td>
<td>Luxembourg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>Czechoslovakia</td>
<td>LV</td>
<td>Latvia</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CZ</td>
<td>Czech Republic</td>
<td>MC</td>
<td>Monaco</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td>Germany</td>
<td>MG</td>
<td>Madagascar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DK</td>
<td>Denmark</td>
<td>ML</td>
<td>Mali</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td>Spain</td>
<td>MN</td>
<td>Mongolia</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI</td>
<td>Finland</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.
Description

METHOD AND SYSTEM FOR INTERFACING TO A TYPE LIBRARY

Technical Field

This invention relates generally to a computer method and system for storing type information, and in particular, a method and system for storing type information in a language-independent manner.

Background of the Invention

The use of object-oriented programming techniques can facilitate the development of complex computer programs. Programming languages that support object-oriented techniques have been developed. One such programming language is C++. The C++ programming language is described in "The C++ Language Reference," published by Microsoft Corporation, which is hereby incorporated by reference.

Two common characteristics of object-oriented programming languages are support for data encapsulation and data type inheritance. Data encapsulation refers to the coupling of data to the functions that operate on the data. Inheritance refers to the ability to define a data type as an extension of other data types. An instance of a data type that couples data and functions is referred to as an "object."

An object is a run time structure for a user-defined type (a class type) that may contain data members and function members. Each class has a class definition that specifies its data members and function members. Each object is declared to be a certain class type. In a preferred embodiment, the run time data structures of an object conforms to the model defined in U.S. Patent Application Serial No. 07/682,537, entitled "A Method for Implementing Virtual Functions and Virtual Bases in a
Compiler for an Object-Oriented Programming Language," which is hereby incorporated by reference.

It is often useful for two computer programs to access the same instance of an object. For example, a first computer program instantiates an object in shared memory and notifies a second computer program of the location of the object. The second computer program then accesses the object in shared memory. However, to access the object, the second computer program needs to be written and compiled with the class definition of the object. Without the class definition, the second computer program would not know how to access the data members and function members of the object. Thus, a computer program is written and compiled with the class definition for each object it wishes to share with another computer program.

When the developer of a computer program wishes to allow other computer programs to access its objects, the developer publishes the class definitions of the objects. The developers of other computer programs could then incorporate the published class definitions into their computer programs. One skilled in the art would appreciate that this publication may use a "header" file containing the class definitions. The developers of other computer programs could then write their computer programs to access objects defined in the header file and "include" the header file in the source code for the computer programs. The source code is then compiled and is ready to access the object.

Because prior methods require that a class definition be available when developing a computer program, there has been limited sharing of objects between computer programs developed by independent developers. This limited sharing occurs because, for example, it is difficult to distribute class definitions to independent developers, who may be located throughout the world, in a timely manner. Similarly, it is difficult to distribute updates to the class definition and difficult for the
developers to integrate these updates and distribute updated versions of the computer programs to users. These difficulties are compounded when a computer program incorporates class definitions provided by several developers. Also, because there is no standard programming language, it can be difficult to adapt a class definition in one programming language to another.

Summary of the Invention

It is an object of the present invention to provide a system for storing and accessing information defining user-defined types.

It is another object of the present invention to provide a system for storing and accessing information for binding to objects.

It is another object of the present invention to provide a system for storing and accessing information for loading class definitions.

It is another object of the present invention to provide a type library interface and implementation for storing and accessing, defining, binding, and loading information relating to objects such that objects instantiated by one computer program can be accessed at run time by a second computer program without need for the second computer program to be compiled with the class definitions at compile time.

It is another object of the present invention to provide language-independent, implementation-independent precompiled type definitions.

These and other objects, which will become apparent as the invention is more fully described below, are obtained by an improved method and system for interfacing to type libraries that store type information. In a preferred embodiment, the present invention defines an interface to a type library. The interface has a plurality of interface methods through which type information can be stored in and retrieved from the type
library. A plurality of implementations of the defined interface are provided. A computer program is compiled using the defined interfaces to access the type information. When the compiled computer program is executed, it accesses the type information using one of the plurality of implementations. In a preferred embodiment, a type library contains type information to allow a compiler to bind to an instance of type at compile time (early binding).

10

Brief Description of the Drawings

Figure 1 is a screen layout showing a sample computer program and application that use a type library.

Figure 2 is a block diagram showing the generation of a type library.

Figure 3 is a block diagram showing sample components used when a word processor invokes a macro interpreter.

Figures 4 through 7 are block diagrams illustrating a preferred type library interface hierarchy.

Figure 5 is a block diagram showing the hierarchy of an ITypeMembers object.

Figure 6 is a block diagram showing the hierarchy of an IFuncDesc object.

Figure 7 is a block diagram showing the hierarchy of an ITypeDesc object.

Figure 8 is a diagram showing steps a macro interpreter executes when interpreting the macro of Code Table 1.

30 Figure 9 is a flow diagram of the routine ExecuteGlobalFunction.

Figure 10 is a flow diagram of the routine GetAddressOfGlobalFunction.

Figure 11 is a flow diagram of the routine ExecuteFunctionMember.

Figure 12 is a flow diagram of the routine GetOffsetOfFunctionMember.
Figure 13 is a block diagram showing the general structure of an interface object.

**Detailed Description of the Invention**

The present invention provides a method and system for a computer program to make its objects (data members and function members) accessible to other computer programs that are developed independently of the computer program. In a preferred embodiment, information defining the type of the objects that are accessible is stored in a "type library." A type library "exposes" to the applications the accessible objects of the computer program. A type library also contains binding information about the exposed objects. The binding information includes information needed by applications, such as compilers or interpreters, to bind to data and function members of the exposed objects. Binding information includes the offsets of data members within instance data and offsets of pointers to function members within virtual function tables. A type library preferably also contains loading information. The loading information includes information needed by applications, such as loaders, to determine the address of global functions and variables. A type library also preferably contains documentation describing the types defined in the library, which alternatively may include references to a help file containing more detailed information. An application accesses the objects of the computer program based on information in a type library.

The present invention provides interfaces (defined below) to a type library through which a developer of a computer program stores type definitions and through which an application retrieves the type definitions. Type definitions, binding information, and loading information for objects to be exposed are stored by instantiating type library interface objects and invoking the function members. An application accesses
the exposed objects by instantiating type library interface objects and invoking the function members. The term "application" refers to code that is exposed to the computer program objects. The code can either execute in the same process or in a separate process from the computer program.

An interface is an abstract class which contains only pure virtual function members, that is, the class has no data members and no non-virtual function members. The function members of an interface are typically semantically related. Table 12 illustrates an interface for the class ISample.

**TABLE 12**

```cpp
15 class ISample
16   {
17     public:virtual RETCODE func0() = 0;
18     virtual RETCODE func1() = 0;
19     virtual RETCODE func2() = 0;
20   }
```

An interface provides an implementation-independent mechanism defining access to an instance of an implementation of the interface, referred to as an interface object. An interface object is a data structure that provides access to the function members of the interface. Figure 13 is a block diagram showing the general structure of an interface object. An interface object comprises instance data 1301, virtual function table 1302, and functions 1303, 1304. The instance data 1301 contains a pointer to the virtual function table and private data that is accessible only to the function members of the interface object. The virtual function table 1302 contains the address of each virtual function defined by the interface. The functions 1303, 1304 contain the code that implements the interface.

The following example illustrates the generation of a type library and the use of the type library by an application. Figure 1 is a screen layout showing a sample computer program and application that use a type library. The computer program is a word processor that interacts
with a user through window 101. The word processor has opened a document named "example.doc" that contains several paragraphs 102. The word processor has menu 103 for invoking a macro interpreter (an application). In this example, the macro interpreter is designed to run as part of the same process of the invoking computer program. The macro interpreter inputs statements in a C++-like language from a user and interprets the statements. When the user selects the macro menu 103, macro window 104 is created. The user inputs the macro statements 105 and then selects the run menu of the macro window 104 to interpret the macro statements. In this example, the macro statements 105 cause each paragraph in the opened document to be right justified.

Continuing with the example, the developer of the word processor publishes a type library containing the type definitions, binding information, loading information, and documentation with the executable code of the word processor. The developer of the macro interpreter publishes a description of its C++-like language. The word processor and the macro interpreter are developed with no special knowledge of the other. That is, the word processor is developed knowing only that it needs to load a macro interpreter stored in a file of a predefined name and invoke that macro interpreter when the user selects its macro menu. Similarly, the macro interpreter is developed knowing only that it needs to look to the type library that it is passed when invoked to access the exposed objects of the invoking computer program. The same macro interpreter can be invoked by other kinds of computer programs, such as a spreadsheet or database program. A user uses the documentation of the type definitions of the exposed word processor objects and the published definition of the language of the macro interpreter to develop macros for the word processor. The macro interpreter inputs these macros and interacts with
the word processor based on the information in the word processor type library to interpret the macros.

Figure 2 is a block diagram showing the generation of a type library. To generate a type library, a "make type library" program 202 inputs a definition 201 of the type of objects that the word processor exposes along with loading information contained in a dynamic link library 205 and uses the type library interface 203 to generate the type library 204. The dynamic link library 205 contains the compiled global functions and the function members that are defined in the definition 201. The make type library program 202 adds the object type definitions to the type library and adds loading information derived from the dynamic link library 205. The program then "compiles" the definitions to generate binding information.

Figure 3 is a block diagram showing the components used when the word processor invokes the macro interpreter. When a user selects the macro menu of the word processor 301, the word processor loads the macro interpreter code 302 in the address space of the word processor 301. In an alternate embodiment, the macro interpreter 302 could be loaded as a separate process and could use standard interprocess communication mechanisms to communicate with the word processor 301. Once the macro interpreter is invoked, the user then inputs macro statements 303. The macro interpreter 302 instantiates type library interface objects 304 to access the information in the word processor type library 305. In this example, the word processor 301 exposes objects of type "document" class and "paragraph" class. In this example, the document object 310 contains instance data 311 and a virtual function table 312. The virtual function table 312 contains pointers to the code that implements the virtual function members of the document class (e.g. GetPara() 313). The function member GetPara returns to the caller a pointer to the paragraph of the
designated index within the open document. The instance data 311 contains a pointer to virtual function table 312 and a pointer to a list of paragraphs 314 in the document. The paragraphs are sequentially ordered by index starting at 0. The list of paragraphs 314 contains pointers to the paragraph objects 320, 330, which contain the text and properties (e.g., justification) of the paragraphs. The word processor instantiates a paragraph object for each paragraph in the document. Each paragraph object contains instance data 321 and virtual function table 322. The virtual function table 322 contains pointers to the code that implements the virtual function members of the paragraph class (e.g. SetJustify() 323). The function member SetJustify sets the paragraph to the designated justification. The word processor also exposes global function GetDoc 340. As its name suggests, the function GetDoc returns to the caller a pointer to the document object for the currently open document.

<table>
<thead>
<tr>
<th>CODE TABLE 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro Justify</td>
</tr>
<tr>
<td>1 Document *pdoc</td>
</tr>
<tr>
<td>2 Paragraph *ppara</td>
</tr>
<tr>
<td>3 pdoc = GetDoc()</td>
</tr>
<tr>
<td>4 i = 0</td>
</tr>
<tr>
<td>5 while (NULL != (ppara = pdoc-&gt;GetPara(i++))</td>
</tr>
<tr>
<td>6 ppara-&gt;SetJustify(&quot;right&quot;)</td>
</tr>
<tr>
<td>EndMacro</td>
</tr>
</tbody>
</table>

Code Table 1 lists macro statements that set the justification of each paragraph in the open document. The numbers to the left of the statements are for reference. In line 1, the macro declares the variable pdoc to point to a document object. In line 2, the macro declares the variable ppara to point to a paragraph object. In line 3, the macro sets the variable pdoc to point to the document object for the open document. The function GetDoc returns the pointer to the document object. In line 4, the macro
sets the variable i to 0. The variable i is used to index through the paragraphs. In lines 5 and 6, the macro loops through each paragraph setting its justification to right justified. The function member GetPara of the document class returns as its value a pointer to the paragraph indexed by its parameter. If the parameter is greater than the number of paragraphs in the open document, then the function return value is set to NULL. The function member SetJustify of the paragraph class sets the paragraph to right justified. The steps that the macro interpreter performs to interpret the macro of Code Table 1 are described in detail below.

Type libraries can also be used to allow an application to create instances of objects defined in the type libraries. For example, a type library can provide a function CreateInstance that creates an instance of a designated type and returns a pointer to the instantiated object. The application can then access the object. The macro interpreter of the above example could instantiate an object a type paragraph, store data in the object using its methods, and use a method of the document object to add the paragraph object to the document.

**Type Library Interfaces**

The type library interfaces provide a mechanism for defining and accessing object type information, binding information, and loading information. In a preferred embodiment, the type library interfaces include the following interfaces:

- ITypeLib
- ITypeInfo
- ITypeBind
- ITypeFixups
- ITypeMembers
- IVarInfo
- ITypeDesc
- IFuncInfo
IFuncDesc
IParamInfo

An implementation of each type library interface has a corresponding interface object. The type library interfaces are defined in a hierarchy.

Figures 4 through 7 are block diagrams illustrating the type library interfaces hierarchy. One skilled in the art would appreciate that the illustrated hierarchy represents a data structure layout of the interface objects in one implementation. Other data structure layouts could also be used. (Referring to Figure 4, template interface object 401 represents the interface object layout used in Figures 4 through 7. Each interface object contains a virtual function table pointer 412, which points to the virtual function table containing pointers to the function members of the interface, and contains private data members 413.) At the root of the type library interfaces hierarchy is the ITypeLib interface object 402. Each type library has one ITypeLib interface object. (In the following, the term "ITypeLib interface object" is shortened to "ITypeLib object"). The ITypeLib object 402 contains a pointer to a list of pointers 403 to the object type definitions contained in the type library. The object type definitions are sequentially ordered starting from 0. Each pointer in the list points to an ITypeInfo object 404, 405. Each ITypeInfo object contains the definition and binding and loading information of one object type. Each ITypeInfo object contains a pointer to an ITypeMembers object 406, an ITypeBind object 407, and an ITypeFixups object 408.

Figure 5 is a block diagram showing the hierarchy below an ITypeMembers object. Each ITypeMembers object contains the definitions of the data members, function members, and base members of the container object type. The container object type for an interface object refers to the object type defined in the ITypeInfo object from which the interface object descends. An ITypeMembers
object 501 contains pointers to three lists of pointers 510, 520, and 530. The list of pointers 510 contains pointers to the function member definitions of the container object type. The list of pointers 520 contains pointers to the data member definitions of the container object type. The list of pointers 530 contains pointers to the base class definitions of the container object type. In one embodiment, the present invention supports multiple inheritance. The function member definitions, data member definitions, and base member definitions are each sequentially ordered starting from 0. Each pointer to a function member definition points to an IFuncInfo object 511, 512. Each IFuncInfo object contains the definition of one function member of the container object type. Each IFuncInfo object contains a pointer to an IFuncDesc object 513, which defines the formal parameters of the function member. Each pointer to a data member definition points to an IVarInfo object 521, 522. Each IVarInfo object contains a definition of a data member or a base member. Each IVarInfo object 521, 522 contains a pointer to an ITypeDesc object 523, which defines the type of a data member. Each pointer to a base member definition points to an ITypeInfo object 531, 532, which define the type of the base members.

Figure 6 is a block diagram showing the hierarchy below an IFuncDesc object. An IFuncDesc object 601 contains a pointer to an ITypeDesc object 602, which defines the return type of the function member; a pointer to a list of pointers 603 to the formal parameters of a function member; and a pointer to an ITypeDesc object 607, which defines the type of THIS pointer for the function member. Each IParamInfo object 604, 605 points to an ITypeDesc object 606, which defines the type of the formal parameter.

Figure 7 is a block diagram showing the hierarchy below an ITypeDesc object. The ITypeDesc objects define the types of various components of an
object type (e.g., data members, formal parameters, base members). An ITypedesc object 701 optionally contains
pointers to an ITypedesc object 702, which defines the
type of the basis (described below) of the component; to
an IFuncDesc object 703, which defines the type of the
function to which the component points; and to an
ITypeInfo object 704, which defines a user-defined type.

Tables 1 through 11 define the type library
interfaces. Each of the methods interfaces are described
in detail in the following. The data members of an
interface object may be stored and retrieved using
methods. The methods that retrieve data are prefixed by
"get," and the methods that store data are prefixed by
"Set." For example, the name of a type library is stored
by the method ITypeLib::SetName, and is retrieved by the
method ITypeLib::GetsetName. The parameters of the methods
are named and typed in a manner to suggest the information
passed as a parameter.

20

class ITypedesc(
  virtual UINT GetTypeinfoCount() = 0;
  virtual SCODE GetTypeinfo(UINT index, ITypeinfo** lplpTypeInfo) = 0;
  virtual SCODE GetDocumentation(BSTR* lpstrDoc) = 0;
  virtual SCODE GetHelpFileName(BSTR* lpstrFileName) = 0;
  virtual DWORD GetHelpContext() = 0;
  virtual SCODE GetTypeBind(ITypedesc** lplpTypeBind) = 0;
  virtual SCODE GetIndexOfName(LPSTR szName, WORD* pWinside) = 0;
  virtual SCODE GetFunctionId(LPSTR szFunctionName, BSTR* lpstrFunctionId) = 0;
  virtual UINT GetFunctionIdCount() = 0;
  virtual SCODE GetFunctionIdOfIndex(UINT index, BSTR* lpstrFunctionId) = 0;
  virtual SCODE GetName(BSTR* lpstrName) = 0;
  virtual WORD GetLanguageCode() = 0;
  virtual SCODE GetTypeCode(UINT index BSTR* lpstrName) = 0;
  virtual SCODE Get Documentation(UINT index, BSTR* lpstrDoc) = 0;
  virtual SCODE SetDocumentation(LPSTR lpstrDoc) = 0;
  virtual SCODE SetHelpFileName(LPSTR lpstrFileName) = 0;
  virtual SCODE SetHelpContext(DWORD dwHelpContext) = 0;
  virtual SCODE SetName(LPSTR szName) = 0;
  virtual SCODE SetLanguageCode(WORD wLangCode) = 0;
  virtual SCODE AddTypeInfo(ITypeInfo* lplpTypeInfo) = 0;
  virtual SCODE RemTypeInfo(UINT i) = 0;
);
ITypeLib Interface

The ITypeLib interface defines the methods of an ITypeLib object. An application uses a function OpenTypeLibrary to open a type library of a designated name, to instantiate an ITypeLib object corresponding to the open type library, and to return a pointer to the instantiated ITypeLib object. The methods of an ITypeLib object provide the functionality to add and remove ITypeInfo objects, to store and retrieve descriptive information of the type library (e.g., help file, name of type library), to retrieve the ITypeInfo objects within the type library, and to retrieve binding and loading information for functions and data defined in the type library. Table 1 lists the ITypeLib interface.

In a preferred embodiment, a type library can store information defining a C++-like class, C++-like enumerators, and global functions and data.

ITypeLib::GetTypeInfoCount

The method GetTypeInfoCount retrieves the number of ITypeInfo objects contained in this ITypeLib object. (In the following, the term "this" before an interface object refers to the instance for which the method is invoked.) The method returns the number as its value.

The number corresponds to the number of pointers in the list of object type definitions 403.

ITypeLib::GetTypeInfo

The method GetTypeInfo retrieves the ITypeInfo object of the designated index for this ITypeLib object. (In the following, the term "designated" refers to a parameter of the method.) The method returns a pointer to the retrieved ITypeInfo object as a parameter.

ITypeLib::GetDocumentation

The method GetDocumentation retrieves a string containing a brief description of this ITypeLib object.
The string contains a description of this ITypeLib object and would typically be used by a "browser." A browser refers to an application that displays object type definitions within a type library to a user. For example, the macro interpreter of the above example could have a browsing mode in which it displayed the object type definitions of the word processor using the word processor or type library. The method returns a pointer to the retrieved string as a parameter.

**ITypeLib::GetHelpFileName**

The method GetHelpFileName retrieves a string containing the file name of a help file for this ITypeLib object. The help file contains information useful to provide assistance to users. The method returns a pointer to the retrieved string as a parameter.

**ITypeLib::GetHelpContext**

The method GetHelpContext retrieves the context for the help file for this ITypeLib object. The help context indicates a location within the help file of pertinent information. The method returns the retrieved help context as a parameter.

**ITypeLib::GetTypeBind**

The method GetTypeBind retrieves an ITypeBind object for this ITypeLib object. The method returns a pointer to the retrieved ITypeBind object as a parameter. The retrieved ITypeBind object allows an application to retrieve binding information for all global functions and global data within this ITypeLib object. Although the functionality of the retrieved ITypeBind object can be provided by instantiating all ITypeInfo objects within this ITypeLib object that have a type kind of MODULE (see below), this method allows a more efficient mechanism to be implemented.
**ITypeLib::GetIndexOfName**

The method GetIndexOfName retrieves the index of the ITypeInfo object of the designated name for this ITypeLib object. The method returns the retrieved index as a parameter.

**ITypeLib::GetName**

The method GetName retrieves the name of this ITypeLib object. The method returns the name as a parameter. The name is intended to be used by compilers that support accessing types defined with a type library. For example, the statement "WP:Document *pdoc" may indicate to the compiler that the type--"WP:Document"--is defined in a type library named "WP" as a type name "Document."

**ITypeLib::GetLanguageCode**

The method GetLanguageCode retrieves a language identifier for this ITypeLib object. The language identifier indicates the natural language (e.g., English, French) of the names and text in this ITypeLib object. The method returns the language identifier as its value.

**ITypeLib::GetTypeName**

The method GetName retrieves the name of the ITypeInfo object of the designated index of this ITypeLib object. This method provides a mechanism for retrieving the object type names without loading the ITypeInfo object. The method returns the retrieved name as a parameter.

**ITypeLib::GetTypeDocumentation**

The method GetTypeDocumentation retrieves a string containing a description of the ITypeInfo object of the designated index of this ITypeLib object. This method provides a mechanism for retrieving object type
documentation without loading the ITypeInfo object. The method returns the retrieved string as a parameter.

\texttt{ITypeLib::SetDocumentation}

The method \texttt{SetDocumentation} stores the designated string as the documentation for this ITypeLib object.

\texttt{ITypeLib::SetHelpFileName}

The method \texttt{SetHelpFileName} stores the designated string as the help file name for this ITypeLib object.

\texttt{ITypeLib::SetHelpContext}

The method \texttt{SetHelpContext} stores the designated help context as the help context for this ITypeLib object.

\texttt{ITypeLib::SetName}

The method \texttt{SetName} stores the designated string as the name of this ITypeLib object.

\texttt{ITypeLib::SetLanguageCode}

The method \texttt{SetLanguageCode} stores the designated code as the language code of this ITypeLib object.

\texttt{ITypeLib::AddTypeInfo}

The method \texttt{AddTypeInfo} adds the designated ITypeInfo object with the designated name to this ITypeLib object. An ITypeInfo object is created with the routine \texttt{CreateTypeInfo} (described below).

\texttt{ITypeLib::RemTypeInfo}

The method \texttt{RemTypeInfo} removes the ITypeInfo object of the designated index from this ITypeLib object.

In an alternate embodiment, each global function defined in a type library is assigned a unique function identifier. The ITypeLib interface provides methods
through which information about a global function can be retrieved using the function identifier. Information can be retrieved more efficiently by using the function identifier rather than by using a string containing the function name.

TABLE 2

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCODE GetName(BSTR* lpbstrName) = 0;</td>
<td>Get name of type information.</td>
</tr>
<tr>
<td>virtual SCODE GetTypeMembers(ITypeInfo** lpiptmembers) = 0;</td>
<td>Get type members.</td>
</tr>
<tr>
<td>virtual SCODE GetTypeBind(ITypeInfoBind** lpilpbinder) = 0;</td>
<td>Get type bind.</td>
</tr>
<tr>
<td>virtual SCODE GetTypeFixups(ITypeInfoFixups** lpilpfixed) = 0;</td>
<td>Get type fixups.</td>
</tr>
<tr>
<td>virtual TYPEKIND GetTypeKind() = 0;</td>
<td>Get type kind.</td>
</tr>
<tr>
<td>virtual SCODE CreateInstance(VOID** ppoobject) = 0;</td>
<td>Create instance.</td>
</tr>
<tr>
<td>virtual WORD GetLanguageCode() = 0;</td>
<td>Get language code.</td>
</tr>
<tr>
<td>virtual SCODE GetDocumentation(BSTR* lpbstrDoc) = 0;</td>
<td>Get documentation.</td>
</tr>
<tr>
<td>virtual SCODE GetHelpFileName(BSTR* lpbstrFileName) = 0;</td>
<td>Get help file name.</td>
</tr>
<tr>
<td>virtual DWORD GetHelpContext() = 0;</td>
<td>Get help context.</td>
</tr>
<tr>
<td>virtual ALIGNKIND GetAlignmentKind() = 0;</td>
<td>Get alignment kind.</td>
</tr>
<tr>
<td>virtual BOOL IsModifiable() = 0;</td>
<td>Is modifiable.</td>
</tr>
<tr>
<td>virtual SCODE GetName(LPSTR lpstrName) = 0;</td>
<td>Get string name.</td>
</tr>
<tr>
<td>virtual SCODE GetTypeKind(TYPEKIND typekind) = 0;</td>
<td>Get type kind.</td>
</tr>
<tr>
<td>virtual SCODE SetDocumentation(LPSTR lpstrDoc) = 0;</td>
<td>Set documentation.</td>
</tr>
<tr>
<td>virtual SCODE SetHelpContext(DWORD dwHelpContext) = 0;</td>
<td>Set help context.</td>
</tr>
<tr>
<td>virtual SCODE SetAlignmentKind(ALIGNKIND alignkind) = 0;</td>
<td>Set alignment kind.</td>
</tr>
</tbody>
</table>

ITypeInfo Interface

The ITypeInfo interface defines the methods of an ITypeInfo object. An ITypeInfo object points to an ITypeInfoMembers object, an ITypeInfoBind object, and an ITypeInfoFixups object. Each of these pointed to objects provide a mechanism for different applications to access information needed by the application in an efficient way.

For example, a browser may use an ITypeInfoMembers object to display the definition of the container object type to a user. (A browser is an application that displays information in a type library to a user). A compiler or interpreter may use an ITypeInfoBind object to retrieve the offsets of data members and function members. A loader may use an ITypeInfoFixups object to determine the address of static data members and static function members of the
container object type. Table 2 lists the ITypeInfo
interface.

Each ITypeInfo object has a designated "type
kind." In a preferred embodiment, the type kinds include
ENUM, RECORD, MODULE, CLASS, ALIAS, and UNION. An ENUM
type kind is a C++-like enumerator. A RECORD type kind is
a C++-like class with no function members and no base
members. A MODULE type kind is an aggregation of non-
class functions and data. A CLASS type kind is a C++-like
class. An ALIAS type kind is a C++-like type definition.
A UNION type kind is a C++-like union.

ITypeInfo::GetName

The method GetName retrieves the name of the
object type defined by this ITypeInfo object. The method
returns the name as a parameter.

ITypeInfo::GetTypeMembers

The method GetTypeMembers retrieves the
ITypeMembers object of the object type defined by this
ITypeInfo object. When this method is called initially
after this ITypeInfo object is instantiated, the retrieved
ITypeMembers object is initialized so that its member
lists are empty. The method returns a pointer to the
retrieved ITypeMembers object as a parameter.

ITypeInfo::GetTypeBind

The method GetTypeBind retrieves the ITypeBind
object instance of the object type defined by this
ITypeInfo object. When this method is called initially
after this ITypeInfo object is instantiated, the retrieved
ITypeBind object is uninitialized. The method returns a
pointer to the retrieved ITypeBind object as a parameter.

ITypeInfo::GetTypeFixups

The method GetTypeFixups retrieves the
ITypeFixups object of the object type defined by this
ITypeInfo object. When this method is called initially after this ITypeInfo object is instantiated, the retrieved ITypeFixups object is uninitialized. The method returns a pointer to the retrieved ITypeFixups object as a parameter.

ITypeInfo::GetTypeKind

The method GetTypeKind retrieves the type kind of the object type defined by this ITypeInfo object. The method returns the retrieved type kind as its value.

ITypeInfo::GetLanguageCode

The method GetLanguageCode retrieves a language identifier of the object defined by this ITypeInfo object. The language identifier indicates the natural language (e.g., English, French) of the text in this ITypeInfo object. The method returns the language identifier as its value.

ITypeInfo::CreateInstance

The method CreateInstance creates an instance of an object defined by this ITypeInfo object. This method returns an error if the type kind of the object defined by this ITypeInfo object is not CLASS. The method returns a pointer to the instantiated object.

ITypeInfo::GetDocumentation

The method GetDocumentation retrieves a string containing brief documentation describing the object type defined by this ITypeInfo object. The method returns the retrieved string as a parameter.

ITypeInfo::GetHelpFileName

The method GetHelpFileName retrieves a string containing the file name of a help file for the object type defined by this ITypeInfo object. The method returns the retrieved string as a parameter.
**ITypeInfo::GetHelpContext**

The method GetHelpContext retrieves the help context for the help file for the object type defined by this ITypeInfo object. The method returns the help context as a parameter.

**ITypeInfo::GetAlignmentKind**

The method GetAlignmentKind retrieves the alignment that should be used when instantiating an object of the object type defined by this ITypeInfo object. The method returns the alignment as its value.

**ITypeInfo::IsModifiable**

The method IsModifiable retrieves a flag that indicates whether the object type defined by this ITypeInfo object can be modified. An object type definition is made not modifiable, for example, to prevent users of a type library from modifying the definitions. The method returns the flag as its value.

**ITypeInfo::SetName**

The method SetName stores the designated name as the name of the object type defined by this ITypeInfo object.

**ITypeInfo::SetTypeKind**

The method SetTypeKind stores the designated type kind as the type kind of the object type defined by this ITypeInfo object.

**ITypeInfo::SetDocumentation**

The method SetDocumentation stores the designated string as the documentation of the object type defined by this ITypeInfo object.
ITypeInfo::SetHelpContext

The method SetHelpContext stores the designated help context as a help context of the object type defined by this ITypeInfo object.

ITypeInfo::SetAlignmentKind

The method SetAlignmentKind stores the designated alignment as the alignment for an object of the object type defined by this ITypeInfo object.

TABLE 3

class ITypeMembers{
    virtual SCODE GetTypeInfoContainer(ITypeInfo** lpITypeInfoContainer) = 0;
    virtual UINT GetFuncCount() = 0;
    virtual SCODE GetFunc(UINT index, IFuncInfo** lpIFuncInfoFunc) = 0;
    virtual UINT GetVarCount() = 0;
    virtual SCODE GetVar(UINT index, IVarInfo** lpIVarInfoVar) = 0;
    virtual SCODE GetBase(ITypeInfo** lpITypeInfoBase) = 0;
    virtual VARTYPE GetVarTypeEnumImpl() = 0;
    virtual BOOL IsLaidOut() = 0;
    virtual SCODE GetTypeDescPtr(ITypeDesc** lpITypeDesc) = 0;
    virtual SCODE GetTypeDescAlias(ITypeDesc** lpITypeDesc) = 0;
    virtual SCODE GetMemberInfoOfIIVar(IMEMORY hMem, IIVarInfo** lpIIVarInfo) = 0;
    virtual SCODE AddFunc(UINT index, IFuncInfo* lpIFuncInfoFunc) = 0;
    virtual SCODE AddVar(UINT index, IVarInfo* lpIVarInfoVar) = 0;
    virtual SCODE AddVar(UINT index) = 0;
    virtual SCODE AddVar(UINT index) = 0;
    virtual SCODE SetBase(ITypeInfo* lpTypeInfoBase) = 0;
    virtual SCODE MakeLaidOut() = 0;
    virtual SCODE SetVarTypeEnumImpl(VARTYPE vt) = 0;
    virtual SCODE GetTypeDescPtr(ITypeDesc* lpITypeDesc) = 0;
    virtual SCODE GetTypeDescAlias(ITypeDesc* lpITypeDescAlias) = 0;
};

ITypeMembers Interface

The ITypeMembers interface defines the methods of an ITypeMembers object. As described above, an ITypeMembers object defines the base members, data members, and function members of the container object type. A complete hierarchy of an object type can be generated by recursively processing the type of the base members. Table 3 lists the ITypeMembers interface. The ITypeMembers interface defines a member handle that is used to identify a member within its container object type. The member handle provides a more efficient way of
identifying a member than by using a string containing the member name.

**ITypeMembers::GetTypeInfoContainer**

5 The method GetTypeInfoContainer retrieves the ITypeInfo object which contains this ITypeMembers object. The method returns a pointer to the retrieved ITypeInfo object as a parameter. When an ITypeMembers object is created, its instance data contains an indication of the container ITypeInfo object. The method can be used to retrieve attributes of the type (e.g., its ITypeBind interface).

**ITypeMembers::GetFuncCount**

15 The method GetFuncCount retrieves the number of function members that are defined for the container object type defined by this ITypeMembers object. The method returns the retrieved number as its value.

**ITypeMembers::GetFunc**

20 The method GetFunc retrieves an IFuncInfo object defining the function member with the designated index of the container object type defined by this ITypeMembers object. This method returns a pointer to the retrieved IFuncInfo object as a parameter.

**ITypeMembers::GetVarCount**

The method GetVarCount retrieves the number of data members that are defined for the container object type defined by this ITypeMembers object. The method returns the number as its value.

**ITypeMembers::GetVar**

The method GetVar retrieves an IVarInfo object defining the data member with the designated index of the container object type defined by this ITypeMembers object.
This method returns a pointer to the retrieved IVarInfo object as a parameter.

**ITypeMembers::GetBase**

The method GetBase retrieves a pointer to an ITypeInfo object defining the base member, if any. This method returns a pointer to the ITypeInfo object as a parameter.

**ITypeMembers::GetVarTypeEnumImpl**

The method GetVarTypeEnumImpl retrieves the integral type of enumerators defined by this ITypeMembers object. If the container object type defined by this ITypeMembers object is not an ENUM type kind, then the method returns an error, else the method returns the integral type as its value.

**ITypeMembers::IsLaidOut**

The method IsLaidOut retrieves a flag indicating whether the container object type defined by this ITypeMembers object has been successfully compiled. The compilation process generates the binding information such that applications can use the binding information to access objects of the container object type. The method returns the retrieved flag as its value.

**ITypeMembers::GetTypeDescPvft**

The method GetTypeDescPvft retrieves an ITypeDesc object that defines the type (e.g., constant, near, far, etc) of the virtual function table pointer of the container object type defined by the container object type defined by this ITypeMembers object. If the container object type has no primary virtual function table pointer, then the method sets the pointer to NULL. The method returns the retrieved pointer as a parameter.
ITypeMembers::GetTypeDescAlias

The method GetTypeDescAlias retrieves the ITypeDesc object that defines the type for which the container object type defined by this ITypeMembers object is an alias. If the container object type is not an ALIAS type kind, then the method sets the pointer to NULL. The method returns a pointer to the retrieved ITypeDesc object as a parameter.

ITypeMembers::GetMemberInfoOfHmember

The method GetMemberInfoOfHmember retrieves an IMemberInfo object defining the member with the designated member handle for the container object type defined by this ITypeMembers object. The method returns a pointer to the retrieved IMemberInfo object as a parameter.

ITypeMembers::AddFunc

The method AddFunc adds a copy of the designated IFuncInfo object defining a function member before the designated index in the list of function members of the container object defined by this ITypeMembers object. A new function may be appended by designating an index equal to the number of function members presently defined in the container object type.

ITypeMembers::RemFunc

The method RemFunc removes the IFuncInfo object with the designated index from the list of function members of the container object type defined by this ITypeMembers object. After removal, the indexes of the function members greater than the designated index are decremented.

ITypeMembers::AddVar

The method AddVar adds a copy of the designated IVarInfo object defining a data member before the designated index in the list of data members of the
container object type defined by this ITypedMembers object. A new data member may be appended by designating an index equal to the number of data members presently defined in the container object type.

ITypedMembers::RemVar
The method RemVar removes the IVarInfo object with the designated index from the list of data members of the container object type defined by this ITypedMembers object. After removal, the indexes of the data members greater than the designated index are decremented.

ITypedMembers::SetBase
The method SetBase sets the base class to the designated ITypedInfo.

ITypedMembers::MakeLaidOut
The method MakeLaidOut compiles the container object type defined by this ITypedMembers object.

ITypedMembers::SetVarTypeEnumImpl
The method SetVarTypeEnumImpl stores the integral type of the enumerators of the container object type defined by this ITypedMembers object.

ITypedMembers::SetTypeDescPvft
The method SetTypeDescPvft stores the type of the virtual function table pointer of the container object type defined by this ITypedMembers object.

ITypedMembers::SetTypeDescAlias
The method SetTypeDescAlias stores the object type for which the container object type defined by this ITypedMembers object is an alias.
TABLE 4

class IMemberInfo{
    virtual SCODE GetName(BSTR* lpstrName) = 0;
    virtual HMEMBER GetHMember() = 0;
    virtual SCODE GetDllEntry(BSTR* lpstrFile, DWORD* lpdwDllOrdinal) = 0;
    virtual SCODE GetDocumentation(BSTR* lpstrDoc) = 0;
    virtual DWORD GetHelpContext() = 0;
    virtual SCODE GetTypeInfoContainer(ITypeInfo** lpiptinfo) = 0;
    virtual SCODE SetName(LPSTR szName) = 0;
    virtual SCODE SetHmember(HMEMBER hmemberVar) = 0;
    virtual SCODE SetDllEntry(LPSTR szFile, DWORD dwDllOrdinal) = 0;
    virtual SCODE SetDocumentation(LPSTR szDoc) = 0;
    virtual SCODE SetHelpContext(DWORD dwHelpContext) = 0;
};

IMemberInfo Interface

The IMemberInfo interface defines the methods of an IMemberInfo object. The IMemberInfo interface serves as a base class for the IVarInfo and IFuncInfo interfaces.

The IMemberInfo interface defines characteristics that are common to data members, base members, and function members. Table 4 lists the IMemberInfo interface.

IMemberInfo::GetName

The method GetName retrieves the name of the member of the container object type defined by this IVarInfo or IFuncInfo object. The method returns the retrieved name as a parameter.

IMemberInfo::GetHmember

The method GetHmember produces the member handle for the member of the container object type defined by this IVarInfo or IFuncInfo object.

IMemberInfo::GetDllEntry

The method GetDllEntry retrieves the name of the dynamic link library (DLL) and ordinal in the DLL for the member of the container object type defined by this IVarInfo or IFuncInfo object, when the member has a DLL entry. Static data members and static function members have DLL entries. If the member has no corresponding
entry in a DLL, the method sets the DLL file name to an empty string. The method returns the retrieved DLL file name and ordinal as parameters.

5 IMemberInfo::GetDocumentation
The method GetDocumentation retrieves a string containing a description of the member of the container object type defined by this IVarInfo or IFuncInfo object. The method returns the retrieved string as a parameter.

10 IMemberInfo::GetHelpContext
The method GetHelpContext retrieves the help context for the member of the container object type defined by this IVarInfo or IFuncInfo object. The help context is an index into the help file of the container ITypeInfo object. The help context is used to access detailed description of the member. The method returns the retrieved help context as a parameter.

15 IMemberInfo::GetTypeInfoContainer
The method GetTypeInfoContainer retrieves the ITypeInfo object that contains this IVarInfo or IFuncInfo object. The method returns a pointer to the retrieved ITypeInfo object as a parameter.

20 IMemberInfo::SetName
The method SetName stores the designated name as the name of the member of the container object type defined by this IVarInfo or IFuncInfo object.

25 IMemberInfo::SetHmember
The method SetHmember stores the designated member handle as the member handle of the member of the container object type defined by this IVarInfo or IFuncInfo object.
IMemberInfo::SetDllEntry

The method SetDllEntry stores the designated file name and designated ordinal as the file name and ordinal for the member of the container object type defined by this IVarInfo or IFuncInfo object.

IMemberInfo::SetDocumentation

The method SetDocumentation stores the designated string as documentation for the member of the container object type defined by this IVarInfo or IFuncInfo object.

IMemberInfo::SetHelpContext

The method SetHelpContext stores the help context for the member of the container object type defined by this IVarInfo or IFuncInfo object.

TABLE 5

class IVarInfo : Public IMemberInfo{
    virtual VARKIND GetVarKind() = 0;
    virtual SCODE GetTypeDesc(ITypeDesc** lpItdesc) = 0;
    virtual SCODE GetValue(VARIANT* lpvariant) = 0;
    virtual BOOL IsStatic() = 0;
    virtual BOOL GetOVar(LONG* lpbpVar) = 0;
    virtual SCODE SetTypeDesc(ITypeDesc* lptdesc) = 0;
    virtual SCODE SetValue(VARIANT variant) = 0;
    virtual SCODE SetStatic(BOOL fStatic) = 0;
};

IVarInfo Interface

The IVarInfo interface defines the methods of an IVarInfo object. The IVarInfo interface inherits the IMemberInfo interface and provides a mechanism for defining data members, base members, and enumerators. The term "variable kind" indicates whether an IVarInfo object defines a base member, data member, or enumerator. The term "variable" refers to data member, base member, or enumerator. Table 5 lists the method introduced by the IVarInfo interface.
IVarInfo::GetVarKind

The method GetVarKind retrieves the variable kind that this IVarInfo object defines.

IVarInfo::GetTypeDesc

The method GetTypeDesc retrieves an ITypeDesc object that defines the object type of the variable defined by this IVarInfo object. The method returns a pointer to the retrieved ITypeDesc as a parameter.

IVarInfo::GetVal

The method GetVal retrieves a variant type when the variable defined by this IVarInfo object is a constant data member or an enumerator. The method returns the pointer as an argument. The method returns an error if the variable is not a constant data member or an enumerator.

IVarInfo::IsStatic

The method IsStatic retrieves a flag that indicates whether the variable defined by this IVarInfo object is static. The method returns the retrieved flag as its value. If the type kind of the container object type is MODULE, then the data member is preferably always static.

IVarInfo::GetOVar

The method GetOVar retrieves the offset from the start of the instance data of an object of the container object type for the variable defined by this IVarInfo object. This offset is only applicable to base members and non-constant data members. The method returns the offset as a parameter.

IVarInfo::SetTypeDesc

The method SetTypeDesc stores the type of the variable defined by this IVarInfo object.
IVarInfo::SetVal

The method SetVal stores the designated variant as the constant value for the variable defined by this IVarInfo object.

IVarInfo::SetStatic

The method SetStatic stores the designated flag as the static flag for the variable defined by this IVarInfo object.

TABLE 6

class ITypeDesc{
    virtual VARTYPE GetVarType() = 0;
    virtual SCODE GetTypeDescBasis(ITypeDesc** lplptdesc) = 0;
    virtual BOOL IsConst() = 0;
    virtual BOOL IsVolatile() = 0;
    virtual DWORD GetSize() = 0;
    virtual SCODE GetFuncDesc(IFuncDesc** lplpfdesc) = 0;
    virtual SCODE GetTypeInfo(ITypeInfo** lplpiinfo) = 0;
    virtual SCODE SetVarType(VARTYPE vt) = 0;
    virtual SCODE SetTypeDescBasis(ITypeDesc* lptdesc) = 0;
    virtual SCODE SetConst(BOOL fConst) = 0;
    virtual SCODE SetVolatile(BOOL fVolatile) = 0;
    virtual SCODE SetSize(DWORD cbSizeType) = 0;
    virtual SCODE SetFuncDesc(IFuncDesc* lpfdesc) = 0;
};

ITypeDesc Interface

The ITypeDesc interface defines the methods of an ITypeDesc object. An ITypeDesc object defines the type of variable. In a preferred embodiment, an ITypeDesc object defines a fundamental type (e.g., integral type, floating type), and a derived type. Table 6 lists the ITypeDesc interface. A derived type includes directly derived types (e.g., array of variables or objects, functions, pointers to variables or objects, references to objects, constants, pointers to class members) and composed derived types (e.g., classes, structures, unions).
ITypeDesc::GetVarType

The method GetVarType retrieves the variable type of this ITypedesc object. This method returns the retrieved variable type as its value. The variable type indicates whether this variable is a particular fundamental type, directly derived type, or composed derivative type.

ITypeDesc::GetTypeDescBasis

The method GetTypeDescBasis retrieves an ITypedesc object that defines the derived type of the variable defined by this ITypedesc object. The method returns a pointer to the retrieved ITypedesc as a parameter.

ITypeDesc::IsConst

The method IsConst retrieves a flag that indicates whether the variable defined by this ITypedesc object is a constant. The method returns the retrieved flag as its value.

ITypeDesc::IsVolatile

The method IsVolatile retrieves a flag that indicates whether the variable defined by this ITypedesc object is volatile. The method returns the retrieved flag as its value.

ITypeDesc::GetSize

The method GetSize retrieves the size of the variable defined by this ITypedesc object. The method returns the retrieved size as its value.

ITypeDesc::GetFuncDesc

The method GetFuncDesc retrieves an IFuncDesc object that defines the prototype of the function to which the variable defined by this ITypedesc object points. The method returns a pointer to the retrieved IFuncDesc as a
parameter. If this variable is not a pointer to a function, then the method sets the pointer to NULL.

**ITypeDesc::GetTypeInfo**

The method GetTypeInfo retrieves an ITypeInfo object that defines the object type of the variable defined by this ITypeDesc object. The method returns a pointer to the retrieved ITypeInfo object. If not applicable, the method sets the pointer to NULL.

**ITypeDesc::SetVarType**

The method SetVarType stores the variable type of the variable defined by this ITypeDesc object.

**ITypeDesc::SetTypeDescBasis**

The method SetTypeDescBasis stores the designated ITypeDesc object to describe the derived type of the variable defined by this ITypeDesc object.

**ITypeDesc::SetConst**

The method SetConst stores the designated flag as the constant flag of the variable defined by for this ITypeDesc object.

**ITypeDesc::SetVolatile**

The method SetVolatile stores the designated flag as the volatile flag of the variable defined by this ITypeDesc object.

**ITypeDesc::SetSize**

The method SetSize stores the designated size as the size of the variable defined by this ITypeDesc object. The size is settable only for fixed-length strings. The size can be inferred from all other variable types.
**ITypeDesc::SetFuncDesc**

The method SetFuncDesc stores the designated IFuncDesc to define the prototype of the function pointed to by the variable defined by this ITypeDesc object.

```cpp
class IFuncInfo : Public IMemberInfo{
  virtual SCODE GetFuncDesc(IFuncDesc** lpfdesc) = 0;
  virtual BOOL IsProperty() = 0;
  virtual BOOL IsPure() = 0;
  virtual UINT GetOvft() = 0;
  virtual FUNKIND GetFuncKind() = 0;
  virtual SCODE SetFuncDesc(IFuncDesc* lpfdesc) = 0;
  virtual SCODE SetProperty(BOOL isProperty) = 0;
  virtual SCODE SetPure(BOOL isPure) = 0;
  virtual SCODE SetFuncKind(FUNKIND funcKind) = 0;
};
```

**IFuncInfo Interface**

The IFuncInfo interface defines the methods of an IFuncInfo object. The IFuncInfo interface inherits the IMemberInfo interface and defines the prototype of a function and binding information of the function. Table 7 lists the IFuncInfo prototype.

**IFuncInfo::GetFuncDesc**

The method GetFuncDesc retrieves a pointer to an IFuncDesc object that defines the prototype information of the function defined by this IFuncInfo object. The method returns a pointer to the retrieved IFuncDesc as a parameter.

**IFuncInfo::GetPropKind**

The method GetPropKind retrieves the property kind for the function defined by this IFuncInfo object. The property kind indicates whether this IFuncInfo is a property function, and if it is, the type of property function. A property function is a function that is syntactically invoked in a language in the same way a data member is accessed. The compiler compiles what appears to
be data member access to a function member invocation. A property function allows actions to be performed when the property is set. For example, if a paragraph has a justify property, then when the justify property is set to right, not only is a data member set, but also a method is invoked to update the paragraph if displayed. A property function type is either set or get. The method returns the property kind as its value.

10 IFuncInfo::IsPure
The method IsPure retrieves a flag indicating whether the function defined by this IFuncInfo object is a pure virtual function. The method returns the retrieved flag as its value.

15 IFuncInfo::GetOvft
The method GetOvft retrieves the offset in the virtual function table of the container object type of the function defined by this IFuncInfo object. If this function is not a virtual function, then an indicator is returned. The offset is set during compilation. The method returns the retrieved offset as its value.

IFuncInfo::GetFuncKind
The method GetFuncKind retrieves the function kind (e.g., virtual, static, etc.) of the function defined by this IFuncInfo object. The method returns the retrieved function kind as its value.

30 IFuncInfo::SetFuncDesc
The method SetFuncDesc stores the designated IFuncDesc object as the prototype information of the function defined by this IFuncInfo object.
IFuncInfo::SetPropKind

The method SetPropKind stores the designated property kind as a property kind of the function defined by this IFuncInfo object.

IFuncInfo::SetPure

The method SetPure stores the designated flag as the pure flag of the function defined by this IFuncInfo object.

IFuncInfo::SetFuncKind

The method SetFuncKind stores the designated function kind as the function kind of the function defined by this IFuncInfo object.

TABLE 8

interface IFuncDesc{
    virtual SCODE GetTypeDescResult(ITypeDesc** lptdesc) = 0;
    virtual UINT GetParamCount() = 0;
    virtual SCODE GetParam(UINT index, IParamInfo** lppparaminfo) = 0;
    virtual CALLINGCONVENTION GetCallingConvention() = 0;
    virtual SCODE GetTypeDescThis(ITypeDesc** lpiptdescThis) = 0;
    virtual BOOL HasAnyNumArgs() = 0;
    virtual UINT GetOptionalArgs() = 0;
    virtual SCODE GetTypeDescResult(ITypeDesc lptdesc) = 0;
    virtual SCODE AddParam(UINT index, IParaminfo** lppparaminfo) = 0;
    virtual SCODE RemParam(UINT index) = 0;
    virtual SCODE SetCallingConvention(CALLINGCONVENTION ccFunc) = 0;
    virtual SCODE SetTypeDescThis(ITypeDesc* lptdescThis) = 0;
    virtual SCODE SetAnyNumArgs(BOOL hasAnyNumArgs) = 0;
    virtual SCODE SetOptionalArgs(UINT cOptionalArgs) = 0;
};

IFuncDesc Interface

The IFuncDesc interface defines the methods of an IFuncInfo object. The IFuncDesc interface provides a mechanism to access function prototype definitions. Table 8 lists the IFuncDesc interface.

IFuncDesc::GetTypeDescResult

The method GetTypeDescResult retrieves an ITypeDesc object that defines the return type of the
function defined by this IFuncDesc object. The method returns a pointer to the ITypeDesc object as a parameter.

IFuncDesc::GetParamCount

The method GetParamCount retrieves the number of formal parameters of the function defined by this IFuncDesc object. The method returns the number as its value. The number indicates the minimum number of parameters required by this function. Additional parameters are of type variant.

IFuncDesc::GetParam

The method GetParam retrieves an IParamInfo object that defines the formal parameter of the designated index of the function defined by this IFuncDesc object. The method returns the pointer to the IParamInfo object as a parameter.

IFuncDesc::GetCallingConvention

The method GetCallingConvention retrieves the calling convention (e.g., Pascal, C++) of the function defined by this IFuncDesc object. The method returns the retrieved calling convention as its value.

IFuncDesc::GetTypeDescThis

The method GetTypeDescThis retrieves an ITypeDesc object that defines the type of the THIS pointer (e.g., near, far) of the function defined by this IFuncDesc object. The method returns a pointer to the ITypeDesc object as a parameter. If this function is static, the method sets the pointer to NULL.

IFuncDesc::HasAnyNumArgs

The method HasAnyNumArgs retrieves a flag indicating whether the function defined by this IFuncDesc supports a variable number of parameters. The method returns the retrieved flag as its value.
IFuncDesc::GetOptionalArgs
The method GetOptionalArgs retrieves the number of optional parameters that the function defined by this IFuncDesc object supports. The method returns the retrieved number as its value.

IFuncDesc::SetTypeDescResult
The method SetTypeDescResult stores the designated ITypeDesc object as the return type of the function defined by this IFuncDesc object.

IFuncDesc::AddParam
The method AddParam adds the designated IParamInfo object before the designated index of parameters of the function defined by this IFuncDesc object. A new parameter may be appended by designating an index equal to the number of parameters presently defined for the function.

IFuncDesc::RemParam
The RemParam removes the formal parameter of the designated index from the formal parameter list of the function defined by this IFuncDesc object.

IFuncDesc::SetCallingConvention
The method SetCallingConvention stores the designated calling convention as a calling convention of the function defined by this IFuncDesc object.

IFuncDesc::SetTypeDescThis
The method SetTypeDescThis stores the designated ITypeDesc object to define the type of the THIS pointer of the function defined by this IFuncDesc object.
IFuncDesc::SetAnyNumArgs
The method SetAnyNumArgs stores the designated number as the number of optional parameters of the function defined by this IFuncDesc object. If the HasAnyNumArgs flag is false, the method returns an error. If the last formal parameter is not an array of variants, the method returns an error.

IFuncDesc::SetOptionalArgs
The method SetOptionalArgs stores the designated number as the number of trailing optional parameters. If the last formal parameter is not a variant, the method returns an error.

TABLE 9
class IParamInfo{
    virtual SCODE GetParamName(LPSTR lpstrParamName) = 0;
    virtual SCODE GetDocumentation(BSTR* lpstrDoc) = 0;
    virtual SCODE GetTypeDesc(ITypeDesc** lpipltdesc) = 0;
    virtual SCODE SetParamName(LPSTR szName) = 0;
    virtual SCODE SetDocumentation(LPSTR szDoc) = 0;
    virtual SCODE SetTypeDesc(ITypeDesc* lptdesc) = 0;
};

IParamInfo Interface
The IParamInfo interface defines the methods of an IParamInfo object. An IParamInfo object provides a mechanism for defining formal parameters. Table 9 lists the IParamInfo interface.

IParamInfo::GetParamName
The method GetParamName retrieves the name of the formal parameter defined by this IParamInfo object. The name is used by languages that support a named parameter call syntax (e.g., ParamB=104). The method returns the retrieved name as a parameter.
IParamInfo::GetDocumentation

The method GetDocumentation retrieves a string containing a description of the formal parameter defined by this IParamInfo object. The retrieved string is returned as a parameter.

IParamInfo::GetTypeDesc

The method GetTypeDesc retrieves the ITypeDesc object that defines the type of the formal parameter defined by this IParamInfo object. The method returns a pointer to the retrieved ITypeDesc object as a parameter.

IParamInfo::SetParamName

The method SetParamName stores the designated name as the name of the formal parameter defined by this IParamInfo object.

IParamInfo::SetDocumentation

The method SetDocumentation stores the designated string as the documentation the formal parameter defined by this IParamInfo object.

IParamInfo::SetTypeDesc

The method SetTypeDesc stores the designated ITypeDesc object to define the type of the formal parameter defined by of this IParamInfo object.

TABLE 10

interface ITypeBind{

virtual SCODE GetTypeInfoContainer(ITypeInfo** lplpTypeInfoContainer) = 0;
virtual UINT GetCbsize() = 0;
virtual UINT GetAlignment() = 0;
virtual UINT GetCbsizeVirt() = 0;
virtual SCODE Bind(HGNAM hgram, KIND* lpkind, VOID** lplpInfoGet, IFuncInfo** lplpInfoPut,
         IFuncInfo** lplpInfoRefPut = 0;
virtual SCODE BindProperty(HGNAM hgram, IFuncInfo** lplpTypeInfo, IFuncInfo** lplpTypeInfoBase) = 0;
virtual BOOL CanCast(ITypeInfo* lpitchTypeBase) = 0;
virtual SCODE GetOverriddenFunction(IFuncInfo* lplpInfo, IFuncInfo** lplpInfoOverride) = 0;
};
**ITypeBind Interface**

The ITypeBind interface defines the methods of an ITypeBind object. The ITypeBind interface provides a mechanism for compilers to retrieve information needed to compile using the container object type. Compilers invoke methods of an ITypeBind object to bind to an instance of an object of the type defined in the container object type. In a preferred embodiment, the bind names (member names and type names) are represented as handles (hgnam) to a name table. In this way, bind names can be efficiently identified during binding by the handle rather than by using a string. A compiler converts each bind name to a name handle before invoking the methods of the ITypeBind interface. The type library may supply a global function for this conversion. Table 10 lists the ITypeBind interface.

**ITypeBind::GetTypeInfoContainer**

The method GetTypeInfoContainer retrieves an ITypeInfo object defining the container object type of this ITypeBind object. The method returns a pointer to the retrieved ITypeInfo object as a parameter.

**ITypeBind::GetCbSize**

The GetCbSize method retrieves the size of an object of the object type defined by this ITypeBind object. This size is the size of the instance data. The method returns the retrieved size as its value.

**ITypeBind::GetAlignment**

The method GetAlignment retrieves the alignment for embedded objects of the container object type of this ITypeBind object. The method returns the alignment as its value.
ITypeBind::GetCbSizeVft

The method GetCbSizeVft retrieves the size of the virtual function table for the container object type of this ITYPEBIND object. The method returns the retrieved size as its value.

ITypeBind::Bind

The method Bind retrieves an IVarInfo or IFuncInfo object that defines the member matching the designated name handle within the container type defined by this ITYPEBIND object. The method returns an indication whether the member is a property. If the member is a property, then the BindProperty method should be invoked next. The method returns a pointer to the IVarInfo or IFuncInfo object as a parameter.

ITypeBind::BindProperty

The method BindProperty retrieves an IFuncInfo object defining a get property function and a put property function of the member matching the designated name handle of the container object type of this ITYPEBIND object. The method returns pointers to the IFuncInfo objects.

ITypeBind::BindType

The method BindType retrieves an ITYPEBIND object which defines the type matching the designated type name. This method is invoked on the ITYPEBIND object returned by the method ITypeLib::GetTypeBind to bind to types defined within that library. The method is also used to bind to nested types.

ITypeBind::CanCast

The method CanCast retrieves an ITYPEINFO object defining the base member of the container object type of this ITYPEBIND object. In an alternate embodiment, an array of ITYPEBIND objects is retrieved to accommodate
multiple inheritance. The method returns a pointer to the retrieved ITypeBind instance.

**ITypeBind::GetOverriddenFunction**

The method GetOverriddenFunction retrieves an IFuncInfo object that defines a virtual function member whose prototype is identical to the function defined by the designated IFuncInfo object that defines a function member of the container object type of the this ITypeBind object. The method returns a pointer to the retrieved IFuncInfo object as a parameter.

```
TABLE 11

class ITypeFixups{
    virtual SCODE GetTypeInfoContainer(ITypeInfo** lpiptinfoContainer) = 0;
    virtual UINT AddressOfMember(HMEMBER hmember, VOID** lplpvoid) = 0;
};
```

**ITypeFixups Interface**

The ITypeFixups interface defines the methods of an ITypeFixups object. The ITypeFixups interface provides a mechanism for loaders to retrieve the address of static function members, static data members, and global functions and data. Table 11 lists the ITypeFixups interface.

**ITypeFixups::GetTypeInfoContainer**

The method GetTypeInfoContainer retrieves an ITypeInfo object that defines the container object type of this ITypeFixups object. The method returns a pointer to the retrieved ITypeInfo object as a parameter.

**ITypeFixups::AddressOfMember**

The method AddressOfMember retrieves the address of the designated member of the container object type of this ITypeFixups object. If the designated member is a static function member or a global function, the method loads the code implementing the function from a dynamic
link library. In an alternate embodiment, all static function members and global functions could be loaded when the type library is first opened. Also, the functions could be stored directly in the type library rather than in a separated dynamic link library. The method returns the address as its value.

Auxiliary Function

The present invention provides various auxiliary functions. The auxiliary functions provide a mechanism for an application to open a type library and instantiate various interface objects. For example, the function CreateITypeDesc creates an uninitialized ITypeDesc object and returns a pointer to the object. These functions, except for the function to open a library, can be defined as global functions within the type library.

Continuing Example

Code Tables 2 through 5 list pseudocode that shows the adding of a class definition to a type library. Code Table 2 lists pseudocode for the routine AddClass. The routine AddClass is passed a pointer to an ITypeLib object into which a designated class definition is to be added. In line 2, the routine invokes routine GenerateClassTypeInfo passing the designated class definition. The routine GenerateClassTypeInfo returns an ITypeInfo object that is initialized with the designated class definition. In line 3, the routine adds the ITypeInfo object to the designated ITypeLib object.

```
CODE TABLE 2

1 AddClass(definition, pplib);
2 {GenerateClassTypeInfo(class_definition, ptinfo);
3 pplib->AddTypeInfo(ptinfo);
};
```
Code Table 3 lists pseudocode for the routine GenerateOfClassTypeInfo. The routine GenerateOfClassTypeInfo is passed a class definition (e.g., name, member names), instantiates all objects needed to describe the class, and returns an ITypeInfo object defining the class. In line 2, the routine invokes routine 'CreateTypeInfo to instantiate an uninitialized ITypeInfo object. In line 3, 4, and 5, the routine sets various attributes of the class. Other attributes of the class could be set including documentation, help file, etc. In line 6, the routine invokes method GetTypeMembers to retrieve an uninitialized ITypeMembers object. In lines 7 through 10, 11 through 14, and 15 through 18, the routine adds the definition of data member, function member, and base members to the ITypeMembers object. In the following, the steps of adding the description of data members is described. One skilled in the art would appreciate that the description of adding function members and base members is analogous. In lines 7 through 10, the routine adds the definition of all the data members of the class to the ITypeMembers. In line 7, the routine initializes the variable i, which is used to index the data members. In lines 8 through 10, the routine loop added the definition of each data member. In line 9, the routine invokes the routine GenerateDataInfo passing in the data member definition and the ITypeInfo object. The routine GenerateDataInfo returns an IVarInfo object containing the definition of the data member. In line 10, the routine adds the IVarInfo object to the ITypeMembers object.
CODE TABLE 3

```c
1 GenerateClassTypeInfo (class_definition, ptinfo)
2 {CreateTypeInfo(ptinfo);
3   ptinfo->SetName(name);
4   ptinfo->SetTypeKind(CLASS);
5   ptinfo->SetAlignment(alignment);
6   ptinfo->GetTypeMembers(ptmembers);
7   i = 0;
8   for (each data member)
9       {GenerateDataInfo(datainfo, pvinfo);
10      ptmembers->AddVar(i++, pvinfo);
11      i = 0;
12   for (each function member)
13       {GenerateFuncInfo(funcinfo, pfinfo);
14          ptmembers->AddFunc(i++, pfinfo});
15         i = 0;
16   for (each base class)
17       {GenerateBaseInfo(baseinfo, pvinfo);
18          ptmembers->AddBase(i++, pvinfo});
19   }
20 }
```

Code Table 4 lists pseudocode for the routine GenerateDataInfo. The routine GenerateDataInfo is passed information defining a data member, generates an IVarInfo object defining the data member, and returns the IVarInfo object. In line 2, the routine invokes routine CreateVarInfo, which creates an uninitialized IVarInfo object. In line 5, the routine sets the name of the IVarInfo instance to the data member name. In line 6, the routine invokes routine GenerateTypeDesc to create and initialize an ITypeDesc instance for the designated data member. In line 7, the routine stores the pointer to the ITypeDesc instance in the IVarInfo instance. In lines 8 and 9, the routine sets various attributes of the data member.
CODE TABLE 4

```c
1  GenerateDataInfo(varinfo, pvinfo)
2  {CreateVarInfo(pvinfo);
3    /*
4    set variable kind to data member;
5    pvinfo->SetName(data member name);
6    GenerateTypeDesc(typedesc, ptdesc);
7    pvinfo->SetTypeDesc(ptdesc);
8    if (variable is static) pvinfo->SetStatic(true);
9    if (variable is constant) pvinfo->SetVal(constant value);
10   }
```

Code Table 5 lists pseudocode for the routine GenerateTypeDesc. The routine GenerateTypeDesc is passed a type definition and an ITypeInfo object, creates and initializes an ITypeInfo object, and returns the ITypeInfo object. In line 2, the routine invokes routine CreateTypeDesc to create an uninitialized ITypeInfo object. In line 3, the routine sets a pointer to an ITypeInfo object. In line 4, the routine sets the variable kind of the ITypeInfo object. In lines 4 through 6, if the variable is a derived type, then the routine generates an ITypeInfo object for the derived from type by recursively calling routine GenerateTypeDesc and stores a pointer to the ITypeInfo object for the derived type. In lines 7 through 11, the routine sets various attributes and creates and initializes an IFuncDesc instance when the variable points to a function.
Figure 8 is a diagram showing steps the macro interpreter executes to interpret the macro of Code Table 1. Figure 8 does not represent a flow diagram of the macro interpreter, but rather represents those steps the macro interpreter executes when interpreting the macro of Code Table 1. In step 801, the macro interpreter opens the type library for the word processor. When the type library is opened, a pointer to an ITypeLib object is returned in variable pplib. In step 802, the macro interpreter allocates memory for the variable pdoc, which is a pointer to a document type. In step 803, the macro interpreter allocates memory for the variable ppara which is a pointer to a paragraph type. In step 804, the macro interpreter interprets line 3 of Code Table 1. The macro interpreter invokes its routine ExecuteGlobalFunction passing it a pointer to the open type library, the name of the global function GetDoc, and an indication that there are no parameters. The routine returns as its value a pointer to the document object for the open document in the word processor. In step 805, the macro interpreter sets the variable pdoc to the return value. In step 806, the macro interpreter interprets line 4 of Code Table 1. The macro interpreter sets the variable i to 0. In steps 807 through 810, the macro interpreter interprets lines 5 and 6 of Code Table 1. In step 807, the macro interpreter

```c
CODE TABLE 5

1  GenerateTypeDesc(typedesc, ptdesc);
2  {CreateTypeDesc(ptdesc);
3      ptdesc->SetVarType(variabletype);
4      if (type has basis)
5          {GenerateTypeDesc(typedesc_of_basis,
6              ptdesc_for_basis);
7             ptdesc->SetTypeDescBasis(ptdescforbasis)};
8      if (type is constant) ptdesc->SetConst(true);
9      if (type is volatile) ptdesc->SetVolatile(true);
10     if (type is pointer to function)
11        {GenerateFuncDesc(functionname, pfdesc);
12           ptdesc->SetFuncDesc(pfdesc)};
```
invokes the routine ExecuteMemberFunction passing a pointer to the ITypeLib object for the open library, a pointer to the document object, an indication that the pointer is to the document object type, the name of the function member to execute (GetPara), and an indication of the number of parameters. The routine returns the result of the GetPara function member. The GetPara function member returns a pointer to a paragraph object of the designated index. In step 808, the macro interpreter sets variable ppara equal to the return value. In step 809, if the variable ppara is equal to null, then all the paragraphs have been processed and the macro interpreter is done interpreting the Code Table 1, else the macro interpreter continues at step 810. In step 810, the macro interpreter invokes routine ExecuteFunctionMember passing a pointer to the ITypeLib object for the open library, a pointer to the paragraph object, an indication that the pointer to the paragraph object points to a paragraph, the name of the function member of the paragraph object to execute (SetJustify), and the parameter "right." The function member SetJustify has no return value. The macro interpreter then loops to step 807 to continue execution of the loop of lines 5 and 6 of Code Table 1.

Figure 9 is a flow diagram of the routine ExecuteGlobalFunction. This routine determines the address of a global function, pushes the parameters for that global function, invokes the global function, and returns the value of the global function. This routine is passed a pointer to an ITypeLib, the name of the global function to execute, and the parameters for the function. This routine returns the value of the global function. In step 901, the routine invokes routine GetAddressOfGlobalFunction passing it a pointer to the ITypeLib object, the function name, and the parameters. The routine GetAddressOfGlobalFunction returns the address of the global function of the designated name within the word processing program. In step 902, the routine pushes
the parameters. The step of pushing the parameters would
depend upon the calling convention of the global function.
In step 903, the routine invokes the global function. In
step 904, the routine stores the value of the global
function as the return value and returns itself.

Figure 10 is a flow diagram of the routine
GetAddressOfGlobalFunction. Routine
GetAddressOfGlobalFunction accesses the type library to
determine the address of the designated global function.
The routine is passed a pointer to an ITypeLib object, the
name of the global function, and the parameters to pass to
the global function. The routine returns the address of
the global function. In step 1001, the routine invokes
routine BindToGlobalFunction passing it the pointer to the
ITypeLib object and the name of the global function. The
BindToGlobalFunction routine returns a pointer to an
ITypeBind object for the designated global function. In
step 1002, the routine invokes routine GetFuncInfo passing
it the pointer to the ITypeBind object. The routine
GetFuncInfo returns a pointer to an IFuncInfo object for
the global function. In step 1003, the routine invokes
routine ValidateParameters passing it the parameters and
the pointer to the IFuncInfo object. The ValidateParameters routine determines whether the
designated parameters are consistent with the formal
parameters specified in the IFuncInfo object. In step
1004, the routine invokes routine GetAddress passing it
the pointer to the IFuncInfo object. The routine
GetAddress returns the address corresponding to the
IFuncInfo object. The GetAddressOfGlobalFunction routine
then returns.

Code Table 6 lists pseudocode for the routine
BindToGlobalFunction. The routine BindToGlobalFunction
inputs a pointer an ITypeLib object and the name of a
global function. The routine retrieves an ITypeBind
object for the designated function and returns a pointer
to that object. The routine retrieves the ITypeBind
object by searching through each ITypeInfo object within the type library that has a type kind of MODULE. The routine checks each function within the ITypeInfo object to determine whether it matches the designated name. If it matches the designated name, then the routine retrieves an ITypeBind object for that function and returns. In an alternate embodiment, the method ITypeLib::GetTypeBind() can be used to retrieve an ITypeBind instance through which the ITypeBind object for the global function of the designated name can be retrieved. Also, in an alternate embodiment, the routine BindToGlobalFunction would check to determine whether multiple global functions with the designated name were defined, rather than return when the first global function was found.

```
CODE TABLE 6

1     BindToGlobalFunction (ptlib, fname, pbind)
2     {count = ptlib->GetTypeInfoCount();
3         for (i = 0; i < count; i++)
4             {ptinfo = ptlib->GetTypeInfo(i, ptinfo);
5                 if ptinfo->GetTypeKind() == MODULE
6                     {ptmembers = ptinfo->GetTypeInfoMembers();
7                         fcount = ptmembers->GetTypeInfoCount();
8                             for (j = 0; j < fcount; j++)
9                                 {ptmembers->GetTypeInfo(j, ptinfo);
10                                     pfname = ptinfo->GetName(name);
11                                     if fname == name
12                                         {ptbind = ptinfo->GetTypeInfoBind(&ptbind);
13                                             return});}}}};
```

Code Table 7 lists pseudocode for the routine GetFuncInfo. The routine GetFuncInfo is passed a pointer to an ITypeBind object and the name of a function to bind. The routine returns a pointer to an IFuncInfo object for the designated function.
Code Table 8 lists pseudocode for the routine ValidateParameters. This routine receives a list of parameters and a pointer to an IFuncInfo object. The routine determines if the designated parameters are consistent with the formal parameters of the IFuncInfo instance. The routine returns as its value an indication of whether the parameters match.

```c
CODE TABLE 8
1 ValidateParameters(parameters, pfinfo)
2 {pfinfo->GetFuncDesc(pfdesc);
3 for ( i = 0; i < pfdesc->GetParamCount(); i++)
4 {pfdesc->GetParam(i, &ppinfo);
5 ppinfo->GetTypeDesc(ptdesc);
6 Determine if parameters[i] matches type
7 pointed to by ptdesc
8 if (!match) {return error};
9 }
}
```

Code Table 9 lists pseudocode for the routine GetAddress. The routine receives a pointer to an IFuncInfo object and returns the address of the function defined by that object. The routine retrieves a handle for that member and then uses an ITypeFixups object to retrieve the address.

```c
CODE TABLE 9
1 GetAddress (pfinfo, address)
2 {Hmember = pfinfo->GetHmember();
3 pfinfo->GetTypeInfoContainer(ptinfo);
4 ptinfo->GetTypeFixups(ptfixups);
5 address = ptfixups->AddressOfMember(Hmember);
6 }
```
Figure 11 is a flow diagram of the routine ExecuteFunctionMember. This routine is passed a pointer to an ITypeLib object, a pointer to an object, the type of the object, the name of a function member, and the parameters to pass to the function member. The routine determines the offset of the function member within the virtual function table of the designated object, invokes that function passing it the parameters, and returns the value of the designated function. In step 1101, the routine invokes routine GetOffsetOfFunctionMember passing it the pointer to the ITypeLib object, the name of the designated function, the object type of which the designated function is a member, and the parameters for the designated function. The GetOffsetOfFunctionMember returns the offset of the designated function member within the virtual function table of the designated object type. In step 1103, the routine pushes the parameters. In step 1104, the routine invokes the designated function of the designated object. In step 1105, the routine stores the return value of the designated function. The routine ExecuteFunctionMember then returns.

Figure 12 is a flow diagram of the routine GetOffsetOfFunctionMember. The routine inputs a pointer to an ITypeLib object, the name of a function member, the container object type of the function member, and parameters to be passed to the function member. The routine validates the parameters and returns the offset of the designated function member in the virtual function table of the container object type. In step 1201, the routine invokes routine BindToFunctionMember passing it pointer to the designated ITypeLib object and the designated container object type. The routine returns with a pointer to an ITypeInfo object for the designated container object type. In step 1202, the routine invokes routine GetFuncInfo passing it a pointer to the designated ITypeInfo object and the name of the designated function member. The routine returns a pointer to an IFuncInfo
object for the designated function. In step 1203, the routine invokes the routine ValidateParameters passing it the designated parameters and the pointer to the IFuncInfo object for the designated function member. In step 1205, the routine invokes routine GetOffset passing it the pointer to the IFuncInfo object for the designated function member. The routine GetOffset returns the offset of the designated function member within the virtual function table of the container object type. The routine GetOffset then returns.

Code Table 10 lists pseudocode for the routine BindToFunctionMember. The routine BindToFunctionMember is passed a pointer to an ITypeLib object and the name of the container object type. The routine returns a pointer to an ITypeBind object for the designated container object type.

<table>
<thead>
<tr>
<th>CODE TABLE 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BindToFunctionMember(ptlib, type, ptbind)</td>
</tr>
<tr>
<td>2 {ptlib-&gt;GetIndexOfName(type, index);</td>
</tr>
<tr>
<td>3 ptlib-&gt;GetTypeInfo(index, ptinfo);</td>
</tr>
<tr>
<td>4 ptinfo-&gt;GetTypeBind(ptbind);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Code Table 11 lists pseudocode for the routine GetOffset. The routine GetOffset is passed an IFuncInfo object and returns the offset of the function member defined by the designated IFuncInfo object.

<table>
<thead>
<tr>
<th>CODE TABLE 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GetOffset(pfinfo, offset)</td>
</tr>
<tr>
<td>2 {offset = pfinfo-&gt;GetOvft();</td>
</tr>
<tr>
<td>3 }</td>
</tr>
</tbody>
</table>

One skilled in the art would appreciate that the type library interfaces as specified in this embodiment support classes defined with at most one base class.
Instances of classes that inherit only one base class are preferably laid out with a single virtual function table. The instance contains a pointer to the virtual function table that is a known offset within the object.

In an alternate embodiment, multiple inheritance is supported. To support multiple inheritance, additional methods are included in several interfaces. In particular, the ITypeMembers interface includes methods to add, remove, and get information describing a base class.

The IVarInfo interface includes a method to retrieve the offset (within an instance of an object defined by the container type) of the introducing base class (possibly virtually inherited) of the base class defined by the IVarInfo object. The IFuncInfo interface includes a method to retrieve the offset (within an instance of an object defined by the container type) of the pointer to the virtual function table containing the address of the function defined by the IFuncInfo object. The ITypeBind interface includes a method to provide information on how to cast an instance of an object defined by the container type to an instance of a base class.

Although the present invention has been described in terms of a preferred embodiment, it is not intended that the invention be limited to these embodiments. Modifications within the spirit of the invention will be apparent to those skilled in the art. The scope of the present invention is defined by the claims that follow.
Claims

1. A method in a computer system for maintaining type information, the method comprising the steps of:
   defining a hierarchy of interfaces, each interface having a plurality of interface methods for accessing type information;
   loading an implementation of the interface methods into the computer system;
   receiving type information; and
   storing the retrieved type information using the loaded implementation of the interface methods.

2. The method of claim 1 wherein the step of defining a hierarchy defines a library interface to store information for a plurality of types.

3. The method of claim 2 wherein the step of defining a hierarchy further defines a type information interface for storing information for one type.

4. The method of claim 3 wherein the step of defining a hierarchy further defines a type members interface for storing information for the members of one type.

5. The method of claim 4 wherein the type members include data members, function members, and base classes.

6. A method in a computer system of accessing an object by a client computer program, the object having an object type and being instantiated by a server computer program, the method comprising the steps of:
   generating a type library, wherein the type library contains a definition of the object type and a function that returns a pointer to the instantiated object;
executing the server computer program, wherein the server computer program instantiates an object of the object type;
executing the client computer program, wherein the client computer program,
determines the definition of the object type of the instantiated object using the generated type library;
determines the location of the function to invoke to retrieve the location of the instantiated object using the generated type library; and
accesses the object using the determined definition and location.

7. The method of claim 6 wherein the step of determining the location of the function includes the step of loading the function in the computer system.

8. A method in a computer system of instantiating an object of a desired type, the method comprising the steps of:
   receiving a definition of the desired type;
   storing the received definition in a type library using an implementation of a type library interface; and
during the execution of a computer program,
determining the definition of the desired type from the type library using the implementation of the type library interface; and
instantiating an object according to the determined definition.

9. A method in a computer system for interfacing to a plurality of implementations of a type library, a type library having a plurality of type definitions, the method comprising the steps of:
 defining an interface to a type library, the interface having a plurality of interface methods for accessing type information stored in a type library;
providing a plurality of implementations of the defined interface;
compiling a computer program using the defined interface, the computer program for accessing type definitions in a type library;
selecting one of a plurality of implementations; and
executing the compiled computer program wherein the computer program accesses a type library using the selected implementation.

10. A method in a computer system for accessing type definitions stored in a type library, the method comprising the steps of:
compiling a computer program, the computer program for accessing type definitions stored in a type library through a predefined interface;
loading into the computer system an implementation of the predefined interface, wherein the implementation provides methods for accessing the type definitions stored in the type library; and
executing the computer program wherein the computer program accesses the type definitions in the type library by using the loaded implementation.

11. The method of claim 10 including the step of loading a second implementation of the predefined interface and wherein the executing computer program accesses the type definitions in a second type library without recompiling the computer program.

12. The method of claim 10 including the step of executing a second computer program wherein the second computer program instantiates an instance of a type defined in the type library and the first computer program binds to the instance using the predefined interface.
Macro Justify
Document *pdoc
Paragraph *ppara
pdoc=GetDoc()
i=0
while (NULL!=(ppara=pdoc ->
    GetPara(i++)))
    ppara -> SetJustify(right)
EndMacro
WP Class Definitions

Class Document
{int size;
paragraph *ptr;
...
virtual ptr GetNextPara() { ... };
...
}

Class Paragraph
{int linespacing;
    boolean bold;
    ...
    virtual VOID SetFont(arg.) { ... };
}

Figure 2
Figure 3

Macro Interpreter

Type Library Interface

Macro Statements

WP Type Library

SUBSTITUTE SHEET (RULE 26)
Figure 6

IFuncDesc

THIS Type ITypeDesc

Parameter Type ITypeDesc

List Of Parameters

IParamInfo

Function Return Type ITypeDesc

SUBSTITUTE SHEET (RULE 26)
Start Interpreting

OpenTypeLib ("WP.LIB", ptlib)

Allocate pdoc as pointer to Document

Allocate ppara as pointer to Paragraph

ExecuteGlobal Function (ptlib, "GetDoc", NULL, retval)

set pdoc = retval

i = 0

ExecuteFunctionMember (ptlib, pdoc, "Document", "GetPara", i++, retval)

if ppara = NULL then
  DONE
else
  ExecuteFunctionMember (ptlib, ppara, "Paragraph", "SetJustify", Right, retval)
end if

Figure 8

SUBSTITUTE SHEET (RULE 26)
Figure 11

Execute Function Member

GetOffsetOf FunctionMember

Push Parameters

call *(*(ptr+ offset)

store return value

Return

Figure 12

GetOffsetOf FunctionMember

BindToFunctionMember

GetFuncInfo

ValidateParameters

GetOffset

Return
### A. CLASSIFICATION OF SUBJECT MATTER

<table>
<thead>
<tr>
<th>IPC(5)</th>
<th>US CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>G06F 9/44</td>
<td>395/700</td>
</tr>
</tbody>
</table>

According to International Patent Classification (IPC) or to both national classification and IPC.

### B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

<table>
<thead>
<tr>
<th>U.S.</th>
</tr>
</thead>
<tbody>
<tr>
<td>395/62, 65, 154, 157, 700</td>
</tr>
</tbody>
</table>

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched.

Electronic database consulted during the international search (name of database and, where practicable, search terms used)

**APG, DIALOG**

### C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category*</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>US, A, 5, 093, 914 (Coplen et al.) 03 March, 1991. See column 5 line 61-column 6 line 11, column 6 lines 22 - 43; column 10 line 41 - column 11 line 16, column 12 line 49 - column 13 line 61.</td>
<td>1-12</td>
</tr>
</tbody>
</table>

Further documents are listed in the continuation of Box C.

<table>
<thead>
<tr>
<th>Date of the actual completion of the international search</th>
<th>Date of mailing of the international search report</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 December 1993</td>
<td>10 JAN 1994</td>
</tr>
</tbody>
</table>

Name and mailing address of the ISA/US Commissioner of Patents and Trademarks

**Box PCT**

Washington, D.C. 20231

Facsimile No. NOT APPLICABLE

Authorized officer **KAKALI CHAKI**

Telephone No. (703) 305-9600
<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>US, A, 5,146,593 (Brandle et al.) 08 September 1992. See &quot;Background&quot; and &quot;Summary of the invention&quot; sections, column 3 lines 26-50, column 3 line 59-column 4 line 2.</td>
<td>1-12</td>
</tr>
</tbody>
</table>