Leo P. Treggiari

Development of the Fortran Module Wizard within DIGITAL Visual Fortran

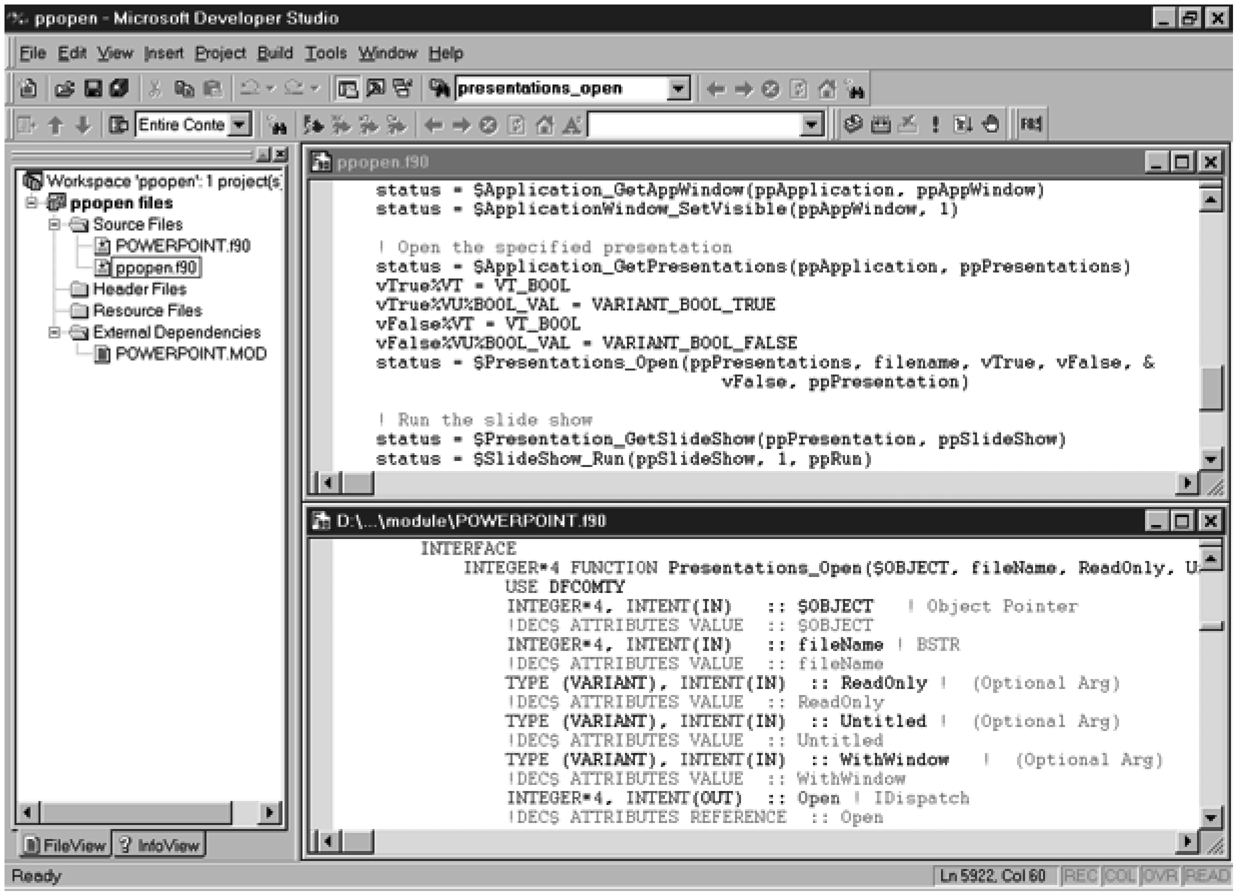
The Fortran Module Wizard is one of the tools in DIGITAL Visual Fortran, a DIGITAL product for the Fortran development environment. Visual Fortran consists of the DIGITAL Fortran 90 compiler and run-time libraries and the Microsoft Developer Studio. Together, these technologies provide a rich set of tools for the Fortran developer who is using the Windows NT and Windows 95 systems. The Fortran Module Wizard generates complete Fortran source code, allowing Fortran applications to invoke routines in a dynamic link library, methods of an Automation object, and member functions of a Component Object Model (COM) object.

DIGITAL Visual Fortran is an integrated development environment for Fortran applications.[1] It is supported on the Windows NT version 4.0 operating system on both Alpha and Intel hardware and on the Windows 95 system. DIGITAL Visual Fortran is a combination of technologies from DIGITAL and Microsoft Corporation. The DIGITAL-supplied compiler and run-time libraries support the DIGITAL Fortran 90 language.[2] DIGITAL Fortran 90 conforms to American National Standard Fortran 90 (ANSI X3.198-1992) and provides many extensions to the Fortran 90 standard. The Microsoft-supplied integrated development environment is the Microsoft Developer Studio, which is also used by Microsoft Visual C++, Microsoft Visual J++ (for Java), other Microsoft tools, and other companies' development tools. Developer Studio includes a text editor, resource editors, project build facilities, an incremental linker, a source code browser, an integrated debugger, and a profiler. The operation of all these tools is controlled from a single application. Figure 1 shows an example of Microsoft Developer Studio from which two Fortran source files are being edited. DIGITAL adds a number of Fortran-specific tools to the environment, one of which is the Fortran Module Wizard.

Design of the Fortran Module Wizard

DIGITAL designed the Fortran Module Wizard to help Fortran developers working in the application-rich Windows environment. The Fortran Module Wizard supports access to dynamic link libraries (DLLs) and servers based upon Microsoft's Component Object Model (COM). This support allows Fortran developers to use the popular mechanisms that make functionality (services) available to other software (clients).

Traditionally, Microsoft and others have provided system interfaces and reusable libraries of code as DLLs. A DLL is a file containing functions that can be called by programs and other DLLs. The role of DLLs on a Windows system is very similar to that of shareable images on the OpenVMS operating system and shared libraries on the UNIX system. Today, DLLs are still the primary mechanism for accessing system interfaces on Windows.



```
    status = $Application_GetAppWindow(ppApplication, ppAppWindow)
    status = $ApplicationWindow_SetVisible(ppAppWindow, 1)

    ! Open the specified presentation
    status = $Application_GetPresentations(ppApplication, ppPresentations)
    vTrue%VT = VT_BOOL
    vTrue%VU%BOOL_VAL = VARIANT_BOOL_TRUE
    vFalse%VT = VT_BOOL
    vFalse%VU%BOOL_VAL = VARIANT_BOOL_FALSE
    status = $Presentations_Open(ppPresentations, filename, vTrue, vFalse, &
                                 vFalse, ppPresentation)

    ! Run the slide show
    status = $Presentation_GetSlideShow(ppPresentation, ppSlideShow)
    status = $SlideShow_Run(ppSlideShow, 1, ppRun)
```

```
        INTERFACE
            INTEGER*4 FUNCTION Presentations_Open($OBJECT, fileName, ReadOnly, U
                USE DFCOMTY
                INTEGER*4, INTENT(IN)    :: $OBJECT   ! Object Pointer
                !DEC$ ATTRIBUTES VALUE   :: $OBJECT
                INTEGER*4, INTENT(IN)    :: fileName ! BSTR
                !DEC$ ATTRIBUTES VALUE   :: fileName
                TYPE (VARIANT), INTENT(IN)  :: ReadOnly !  (Optional Arg)
                !DEC$ ATTRIBUTES VALUE   :: ReadOnly
                TYPE (VARIANT), INTENT(IN)  :: Untitled !  (Optional Arg)
                !DEC$ ATTRIBUTES VALUE   :: Untitled
                TYPE (VARIANT), INTENT(IN)  :: WithWindow   !  (Optional Arg)
                !DEC$ ATTRIBUTES VALUE   :: WithWindow
                INTEGER*4, INTENT(OUT)   :: Open ! IDispatch
                !DEC$ ATTRIBUTES REFERENCE  :: Open
```

Figure 1
Microsoft Developer Studio, Two Fortran Source Files Being Edited

When Microsoft introduced OLE version 1, the name OLE was an acronym for object linking and embedding. OLE version 1 enabled compound documents by allowing a document to link to, or embed data from, another document. In 1993, Microsoft introduced COM as the base architecture of OLE version 2.[3] COM is an extensible architecture that provides mechanisms for creating and using software components. A software component consists of reusable pieces of code and data in binary form that can be plugged into other software components from other vendors with relatively little effort.[4] Like DLLs, COM allows a software developer to provide a set of services to multiple clients. In addition, COM has the advantage of allowing the services to reside in another process and on another machine. (Distributed COM [DCOM] allows objects to be created and used on remote machines.) COM also contains features that aid in the deployment and evolution of the services.[5] Microsoft has extended its languages and tools to aid software developers in the creation of clients and servers based upon COM (hereafter referred to as clients and servers in this paper).

Why does a Fortran developer need help accessing services in DLLs and servers? Calling code that is written in another programming language is, in general, difficult. There are complex issues around calling standards and data type representations. If a mistake is made in manually translating a function signature from one language into another, today's programming environments are of little help. The application can fail at a point in the code, for example in the routine prolog, which does little to suggest the cause of the problem. Often, solving these problems requires understanding the intricacies of calling standards and single stepping through assembly code. Calling the components in a server also requires understanding and properly using a number of COM programming interfaces.

The Fortran Module Wizard deals with the difficulties. It reads a description of a service, which the service provider created, and generates Fortran source code. This automatically generated code makes calling these services as easy as calling another Fortran function or subroutine.

Enabling Technologies

Components of COM, Fortran 90, and the Microsoft Developer Studio enable the functionality of the Fortran Module Wizard. This section gives an overview of these technologies.

COM Technologies

As mentioned earlier, COM provides mechanisms for creating reusable software components. This paper attempts to explain only those parts of COM, and some technologies based on COM, necessary for the reader to understand the use of server functionality from code generated by the Fortran Module Wizard. COM, OLE, and ActiveX, of course, contain many more mechanisms.[6] A number of the references listed at the end of this paper are good sources of further reading.[4–7] Much of the description of COM in the following section is taken from the Component Object Model Specification.[8]

COM Objects COM is an object-based programming model designed to promote software interoperability. In other words, COM allows two or more applications or components to easily cooperate with one another, even if they were written by different vendors at different times, in different programming languages, or if they are running on different machines running different operating systems. COM defines a completely standardized mechanism for creating objects and for clients and objects to communicate. Unlike traditional object-oriented programming environments, these mechanisms are independent of the applications that use object services and of the programming languages used to create the objects. COM therefore defines a binary interoperability standard rather than a language-based interoperability standard on any given operating system and hardware platform.

To support its interoperability features, COM defines and implements mechanisms that allow components to connect to each other as objects. The definition of an object is a piece of software that contains the functions that represent what the object can do (its intelligence) and associated state information for those functions (data). In other words, an object is some data structure and some functions to manipulate that data. In this paper, we use the term object to mean an object instance, as opposed to an object class. An object class is similar to a derived-type in Fortran 90 or a structure in C. It specifies a blueprint for object instances that a server will create upon a client's request. An important principle of object-oriented programming is encapsulation, in which the exact implementation of those functions and the exact format and layout of the data is only of concern to the object itself. This information is hidden from the clients of an object and can therefore be changed without affecting the client.

With COM, components interact with each other and with the system through collections of function calls, also known as methods or member functions or requests, called interfaces. An interface is a semantically related set of member functions. The interface as a whole represents a feature of an object. The member functions of an interface represent the operations that make up the feature.

For a quick look at a simple example of a COM object, imagine a Calculator object that is willing to provide arithmetic services to any client. It could support an interface named ICalculate. By convention, the letter I always prefixes the name of an interface. The ICalculate interface could contain member functions named Add, Subtract, Multiply, Divide, etc. If a client wanted to use the services of the Calculator object, it would request COM to create an object of class Calculator and request the ICalculate interface. It could then call the member functions of the ICalculate interfaces (Add, Subtract, etc.).

With COM, a pointer to an object is actually a pointer to a particular interface that the object supports. All COM objects support the interface named Iunknown, which contains the member functions named AddRef, Release, and QueryInterface. All COM objects must implement these member functions. AddRef and Release implement object reference counting. Clients use them to tell an object when they are using it and when they are done. Objects delete themselves when they are no longer being used by any client. QueryInterface is the basis for a process called interface negotiation, whereby a client asks an object what services it is capable of providing. For example, if a client had a pointer to the Calculator object's IUnknown interface, it could get a pointer to its ICalculate interface by calling the IUnknown QueryInterface member function. In general, an object can support multiple interfaces and a client can use QueryInterface to get a pointer to any of them. Examples in which Fortran code calls member functions in interfaces are given in the section Fortran Module Wizard Functionality. Microsoft defines a number of useful interfaces. Object class creators are free to use existing interfaces and define their own.

Automation Objects One Microsoft-defined interface, IDispatch, is the basis for Automation.[9] Any object that supports this interface, also known as a dispinterface, is an Automation object, and can be accessed by any Automation client. An Automation object exposes methods and properties. Methods are functions that perform an action on an object and are similar to the member functions of COM objects. Properties hold information about the state of an object. A property can be represented by a pair of methods; one for getting the property's current value, and one for setting the property's value.

The capabilities of an Automation object are similar to those of a COM object. An Automation object is, in fact, a COM object; that is, it supports the IUnknown interface as well as the IDispatch interface. However, the mechanisms for using the services of the two are very different. Microsoft designed Automation based on the needs of scripting or macro languages (i.e., Visual Basic). It does not require understanding the intricacies of calling conventions as does COM. It supports mechanisms more suitable to the dynamic querying of an object's capabilities. This makes Automation more suited to late binding of objects, that is, invoking methods of a previously unknown object at run time.

An Automation client accesses all the methods and properties of an Automation object through a single member function of the IDispatch interface named Invoke. The client passes Invoke a number of arguments that identify

- The method, its arguments, and a place to receive the return value, or
- The property and its new value, or
- The property and a place to receive its current value

In fact, Invoke could be described as the Swiss army knife of Automation programming.

Most of the differences between Automation objects and COM objects are hidden by the Fortran interfaces that the Wizard generates.

Object Identification To enable the use of COM objects created by disparate groups of developers, there must be a method of uniquely identifying an object class regardless of its origin. COM uses globally unique identifiers (GUIDs) to do this. A GUID is a 16-byte integer value that is guaranteed (for all practical purposes) to be unique across space and time. COM uses GUIDs to identify object classes, interfaces, and other things that require unique identification. COM provides a routine named CoCreateGUID, and Microsoft provides a utility named GUIDGEN, that a developer uses to generate a GUID. Assigning a GUID to an object class or interface is the job of the creator of the class or interface. To create an instance of an object, the developer needs to tell COM the GUID of the object. Using 16-byte integers for identification is fine for computers, but it poses a challenge for the typical developer. COM supports the use of a less precise, textual name called a programmatic identifier (ProgID). A ProgID takes the form:

```
application_name.object_name.object_version
```

For example, the name of the Basic object of the Microsoft Word application is Word.Basic.1. Similarly, interfaces are usually discussed using their Ixxx name (for example, IUnknown), but their GUID uniquely identifies them. ProgIDs are not supplied for all objects. They are normally supplied only for Application objects. An Application object is a top-level object that becomes active when the application starts. It provides a starting point for clients to access all of an application's subordinate objects.

Type Information Type information contains descriptions of object classes, interfaces, DLLs, data structures, and so forth that are independent of any programming language. A developer accesses type information through an interface named ITypeInfo.[7] A client can get a pointer to type information from

- A running Automation object
- A running COM object that supports the IProvideClassInfo interface
- A type library

A type library is a collection of type information for any number of object classes, interfaces, etc. A developer can store a type library in a separate file (using a .TLB extension by convention), or as part of another file. For example, the type library that describes the type information for a DLL can be stored in the .DLL file itself. Since the type information is stored in a file, it is available regardless of whether or not the client has a pointer to the object(s) that the information describes.

The easiest way to create a type library is to write a script in the Microsoft Interface Definition Language (IDL). The Microsoft IDL compiler (MIDL) reads an IDL script and creates a .TLB file.[10] An IDL script is similar to a C++ header file with additional syntax for information required by COM. An example of such information is whether an argument to a member function is an input, an output, or an input/output argument.

To use the Fortran Module Wizard, the developer must know where to find type information for the functionality to be used. Some examples of this are given in the section Fortran Module Wizard Functionality.

Fortran 90

This section describes features of the DIGITAL Fortran 90 language that the Fortran Module Wizard uses in the code that it generates.

Modules Fortran 90 does not support objects, but it does provide a new form of program unit called a module. A Fortran module is a set of declarations that are grouped together under a global name and are made available to other program units by means of the Fortran USE statement. These modules have similarities to C include files but are more powerful.

The Fortran Module Wizard generates a source file containing one or more Fortran modules and places the following types of information in the modules:

- Derived-type definitions—Fortran equivalents of data structures that are found in the type information.

- Procedure interface definitions—Fortran interface blocks that describe the procedures found in the type information.
- Procedure definitions—Fortran functions and subroutines that are wrappers for the procedures found in the type information. The wrappers make the external procedures easier to call from Fortran by handling data conversion and low-level invocation details.

The use of modules allows the Fortran Module Wizard to encapsulate the data structures and procedures exposed by an object or DLL in a single place. These definitions can be shared in multiple Fortran programs.

Attributes The DIGITAL Fortran 90 language supports a number of calling convention attributes that allow Fortran programs to call programs written in other programming languages. Some attributes select the calling convention (STDCALL, C, VARYING). Others determine whether an argument is passed by value or by reference (VALUE, REFERENCE). Another attribute defines the external name of the procedure (ALIAS).

Pointer To Procedure The address of a COM member function is never known at program link time. The developer must get a pointer to an object's interface at run time, and the address of a particular member function is computed from that. We have extended the DIGITAL Fortran 90 language to support a Pointer To procedure.

Microsoft Developer Studio

Microsoft Developer Studio provides a number of methods that allow software developers to extend its environment.[11] This section describes these methods.

Tools Menu Developer Studio contains a Customize dialog box through which the developer can add utilities to the Tools menu and then run those utilities from within Developer Studio.

Gallery The Developer Studio Gallery provides a central repository for all reusable parts of projects. The reusable parts can range from something as simple as a bitmap to something as complex as a DLL.

Developer Studio Object Model Developer Studio provides a set of COM objects that give developers programmatic control of its functionality. Users can create commands that perform specific tasks and add them to a toolbar. The Developer Studio Object Model is programmed in three ways: (1) by creating macros in the Visual Basic Scripting Edition Language (VBScript); (2) by creating a Developer Studio DLL Add-in, which is a server implemented as a DLL; and (3) by creating a separate Automation client that connects to the Developer Studio objects.

Wizards A wizard is code that creates the starter files for a new application or adds a feature to an existing application. Wizards that add features are stored in the Developer Studio Gallery. Wizards that create starter files for a new application are called AppWizards. When the developer requests the creation of a new project, Developer Studio presents a list of the types of project that can be created (for example, a console application or a DLL). In addition, it lists the installed AppWizards that can generate complete applications. Often they contain options that allow the developer to choose the features of a generated application.

Microsoft Visual C++ provides a number of AppWizards; most of them can create typical C++ applications. In addition, to aid developers in extending Developer Studio, one AppWizard creates the starter files for a custom AppWizard, and another creates the starter files for a DLL Add-in. The Fortran Module Wizard is currently implemented as an application that runs from the Developer Studio Tools menu. In the future, it may be a Developer Studio AppWizard.

Fortran Module Wizard Functionality

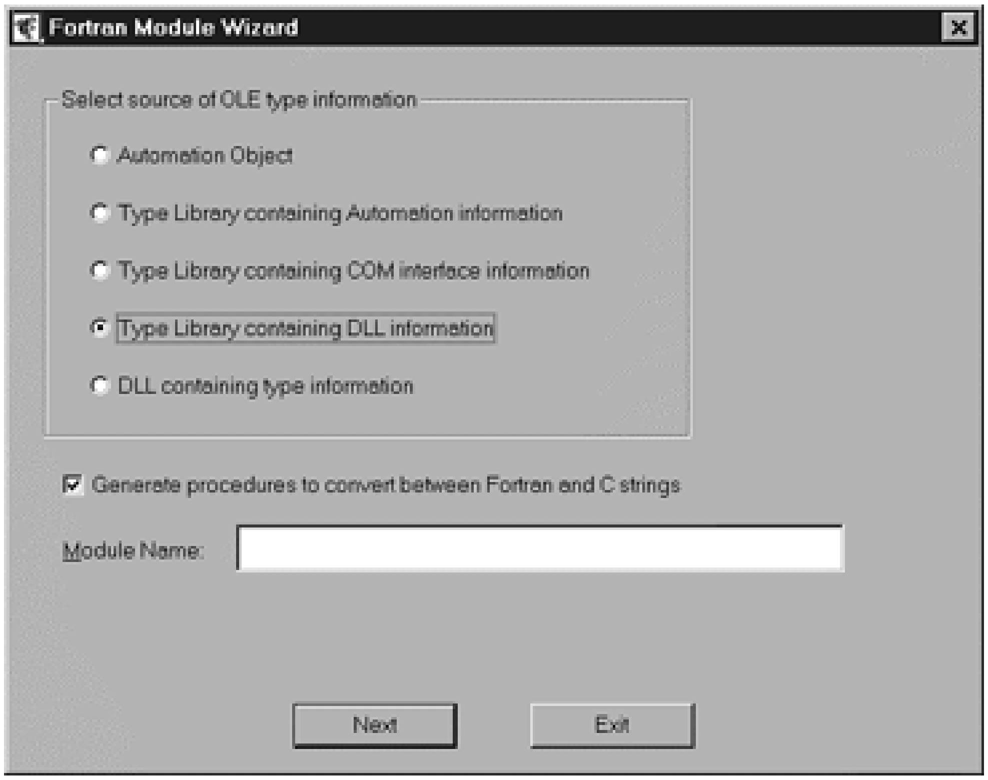
This section describes the user interface of the Fortran Module Wizard and presents some samples of the code generated by the Wizard. It also shows examples of calling the generated code from Fortran.

User Interface

Upon opening the Fortran Module Wizard from the Tools menu, the user is presented with a series of dialog boxes. From these, the user selects the type information for the functionality needed.

Figure 2 shows the first dialog box. It requests the user to choose the source of the type information that describes the required functionality. The developer must consult the documentation to determine what type of object (or DLL) the functionality is implemented as, and where to find its associated type information. The choices are the following:

- Automation object
- Type library containing automation information
- Type library containing COM interface information
- Type library containing DLL information
- DLL containing type information


Fortran Module Wizard

Select source of OLE type information

( ) Automation Object

( ) Type Library containing Automation information

( ) Type Library containing COM interface information

(●) Type Library containing DLL information

( ) DLL containing type information

[✓] Generate procedures to convert between Fortran and C strings

Module Name:

Figure 2
Fortran Module Wizard Dialog Box

Automation Object Microsoft recommends that servers provide a type library. Some applications, for example Microsoft Word version 7.0, do not, but they do provide type information dynamically when running. When this option is selected, Developer Studio displays the dialog box shown in Figure 3. The user then enters the name of the application, the name of the object, and optionally the version number. Note that this method works only for objects that provide a ProgID. ProgIDs are entered into the system registry and identify, among other things, the executable program that is the object's server.

After the user enters the information and presses the "Generate button," the Fortran Module Wizard asks COM to create an instance of the object identified by the ProgID that the Wizard constructs from the user-supplied information. COM starts the object's server if it needs to do so. The Wizard then asks the object for its type information and generates a file containing Fortran modules.

Other Options If the user chooses one of the remaining options, that is, any of the type libraries or the DLL (see Figure 2), Developer Studio displays the dialog box shown in Figure 4. From this dialog box, the user chooses the type library (or file containing the type library) and, optionally, the specific components of the type library.

At the top of the dialog box, a "combo box" lists all the type libraries that have been registered with the system. Their file names have a number of different file extensions, for example, .OLB (object libraries) and .OCX (ActiveX controls). The user either selects a type library from the list or presses the "Browse button" to find the file using the standard "Open dialog box." After selecting a type library, the user presses the "Show button" to list the interfaces described in the type library. By default, the Fortran Module Wizard uses all the interfaces; however, the developer can select the ones desired from the list.

After the user enters the information and presses the "Generate button," the Fortran Module Wizard asks COM to open the type library and generates a file containing Fortran modules.

Generated Code

The Fortran Module Wizard generates different code, depending upon the type of object or DLL described by the type information. Note that the generated code is a static representation of an object's type information. If the type information should change in a future release of the object, the Wizard would need to be run again.

Fortran Run-time Support DIGITAL Visual Fortran provides a set of run-time routines that present to the Fortran programmer a higher-level abstraction of the

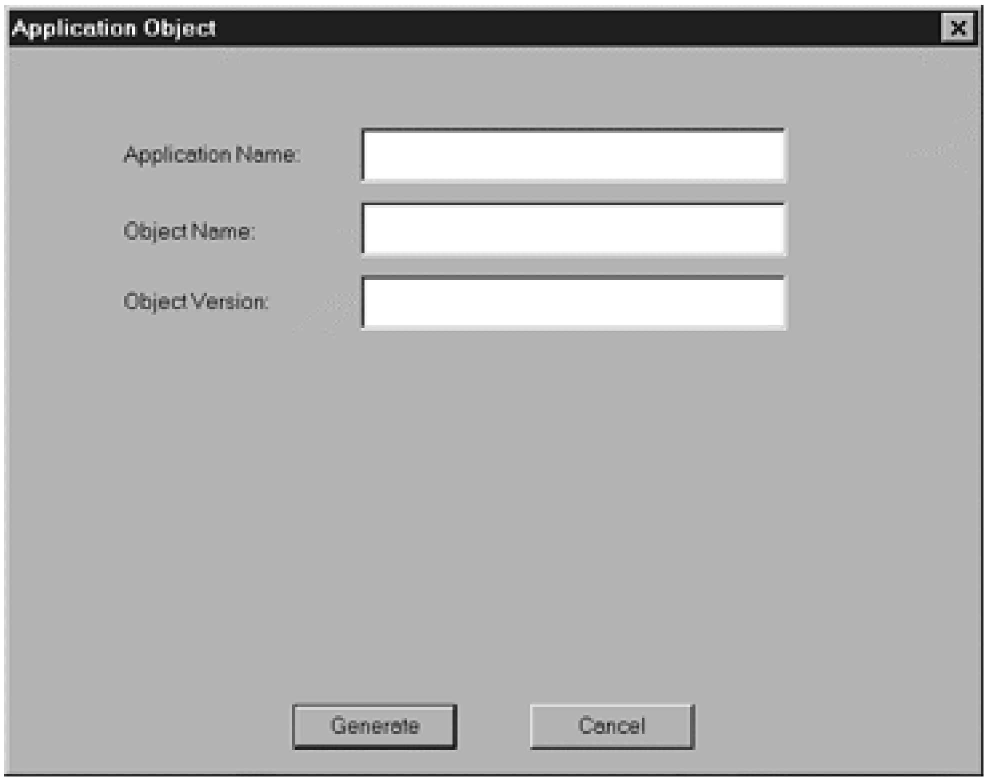
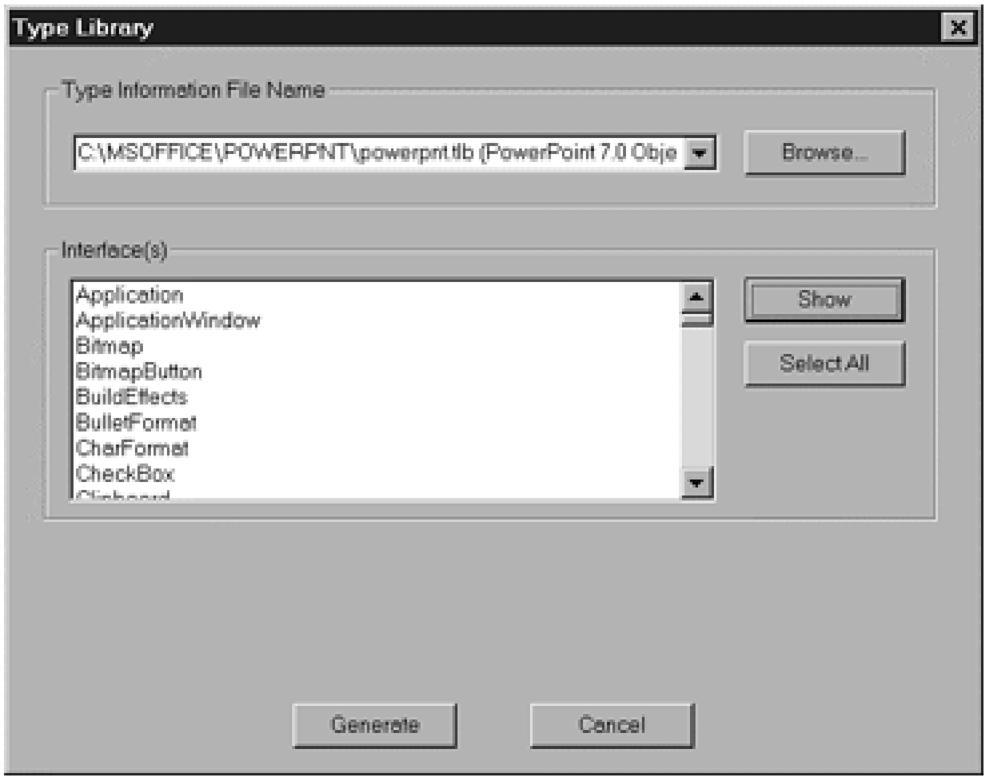


Figure 3
Microsoft Developer Studio Dialog Box for Application Object Selection



C:\MSOFFICE\POWERPNT\powerpnt.tlb (PowerPoint 7.0 Obje

Application
ApplicationWindow
Bitmap
BitmapButton
BuildEffects
BulletFormat
CharFormat
CheckBox

Figure 4
Microsoft Developer Studio Dialog Box for Type Library Selection

IDispatch member functions and other COM functions. The routines are used in the code that the Wizard generates. They allow the programmer to perform the following tasks:

- Initialize the COM library.
  - COMInitialize initializes the COM library.
  - COMUninitialize uninitializes the COM library.
- Get an interface pointer of an object.
  - COMCreateObject passes a programmatic identifier or class identifier, and it creates an instance of an object and returns a pointer to one of the object's interfaces.
  - COMGetActiveObject passes a programmatic identifier or class identifier, and it returns a pointer to an interface of a currently active object.
  - COMGetFileObject passes a file name, and it returns a pointer to the IDispatch interface of an Automation object that can manipulate the file.
  - COMCLSIDFromPROGID passes a programmatic identifier, and it returns the corresponding class identifier.
  - COMCLSIDFromString passes a class identifier string, and it returns the corresponding class identifier.
- Get or set the value of a property of an Automation object.
  - AUTOSetProperty passes the name or identifier of the property and a value, and it sets the value of the Automation object's property.
  - AUTOGetProperty passes the name or identifier of the property, and it gets the value of the Automation object's property.
- Invoke a method of an Automation object.
  - AUTOAllocateInvokeArgs allocates an argument list data structure that holds the arguments that the user will pass to AUTOInvoke.
  - AUTOAddArg passes an argument name and value, and it adds the argument to the argument list data structure.
  - AUTOInvoke passes the name or identifier of an object's method and an argument list data structure, and it invokes the method with the passed arguments.
  - AUTODeallocateInvokeArgs deallocates an argument list data structure.
  - AUTOGetExceptionInfo retrieves the exception information when a method has returned an exception status.
- Perform IUnknown interface member functions.
  - COMAddObjectReference adds a reference to an object's interface.
  - COMReleaseObject indicates that the program is done with a reference to an object's interface.
  - COMQueryInterface passes an interface identifier, and it returns a pointer to an object's interface.

DIGITAL Visual Fortran provides three Fortran modules that define basic COM information:

- DFCOMTY defines basic COM types.
- DFCOM defines the interfaces to the DIGITAL Visual Fortran COM routines and to some COM system routines.
- DFAUTO defines the interfaces to the DIGITAL Visual Fortran Automation routines.

Automation Objects Figure 5 contains code generated by the Fortran Module Wizard for the Word.Basic object of Microsoft Word version 7.0. Word.Basic is an Automation object with almost 1,000 methods. These methods represent the functionality of the Word Basic language, which is the programming interface to Microsoft Word. The Microsoft Word, Word Basic documentation contains information on the methods and their arguments.[12] We discuss some of the methods here in a simple example of Fortran code automating Word Basic to perform the task of replacing all the occurrences of a word in a document with another word. The Word.Basic methods of interest for this example are the following:

- AppShow makes the Microsoft Word application visible.
- FileOpen opens a document.
- EditReplace replaces a string with another string.
- FileSaveAs saves a document.

Figure 5 contains code from the Fortran subroutine generated for the Word Basic FileOpen method. It is representative of the code generated for all Automation methods. The lines are annotated on the left side with numbers that are not part of the source code but correspond to the list below. Note that the naming convention used for the generated wrappers is *objectname_methodname*. Any periods in the name are replaced by underscores.

1. If the type information provides a comment that describes the method, the comment is placed before the beginning of the procedure.
2. The first argument to the procedure is always $OBJECT. It is a pointer to an Automation object's IDispatch interface. The last argument to the procedure is always $STATUS. This optional argument can be specified if the Fortran programmer wishes to examine the return status of the method. The IDispatch Invoke member function returns a status of type HRESULT, which is a 32-bit value. HRESULT has the same structure as a Win32 error code. In between the $OBJECT and $STATUS arguments are the method arguments' names determined from the type information. When the type information does not provide a name for an argument, the Fortran Module Wizard creates a $ARGn name.

```
1-  !Opens an existing document or template
2-  SUBROUTINE Word_Basic_FileOpen($OBJECT, Name, ConfirmConversions,
      ReadOnly, LinkToSource, AddToMru, PasswordDoc, PasswordDot,
      Revert, WritePasswordDoc, WritePasswordDot, Connection,
      SQLStatement, SQLStatement1, $STATUS)
    !DEC$ ATTRIBUTES DLLEXPORT     :: Word_Basic_FileOpen
    IMPLICIT NONE
    INTEGER*4, INTENT(IN)          :: $OBJECT   ! Object Pointer
3-  !DEC$ ATTRIBUTES VALUE         :: $OBJECT
4-  CHARACTER*(*), INTENT(IN), OPTIONAL :: Name ! BSTR
    !DEC$ ATTRIBUTES REFERENCE     :: Name

    ...
    INTEGER*4, INTENT(OUT), OPTIONAL :: $STATUS ! Method status
    !DEC$ ATTRIBUTES REFERENCE     :: $STATUS
    INTEGER*4 $$STATUS
    INTEGER*4 invokeargs
5-  invokeargs = AUTOALLOCATEINVOKEARGS()
6-  IF (PRESENT(Name)) CALL AUTOADDARG(invokeargs, 'Name', Name,
                                   .FALSE., VT_BSTR)

    ...
7-  $$STATUS = AUTOINVOKE($OBJECT, 'FileOpen', invokeargs)
8-  IF (PRESENT($STATUS)) $STATUS = $$STATUS
9-  CALL AUTODEALLOCATEINVOKEARGS (invokeargs)
    END SUBROUTINE Word_Basic_FileOpen
```

Figure 5
Representative Code Generated for Automation Methods

3. This is an example of an attribute statement used to specify the calling convention of an argument.
4. Methods can take optional arguments that must follow all the required arguments. In this method, there are no required arguments. The Fortran Module Wizard generates source lines for each argument using the data type and calling conventions found in the type information.
5. AUTOAllocateInvokeArgs allocates a data structure that is used to collect the arguments that the programmer passes to the method. AUTOAddArg adds an argument to this data structure.
6. For each optional argument, the Fortran PRESENT function is used to determine if the caller supplied the argument. If so, the argument is added to the argument list.
7. AUTOInvoke invokes the named method passing the argument list. This returns a status result.
8. If the caller supplied a status argument, the code copies the status result to it.
9. AUTODeallocateInvokeArgs deallocates the memory used by the argument list data structure.

Figure 6 shows code from a user-written Fortran program that invokes Microsoft Word to replace all the occurrences of a word in a document with another word. The example code is annotated with numbers that correspond to the following list.

1. COMCreateObject requests COM to create an object with the ProgID Word.Basic. A pointer to the Word.Basic object's IDispatch interface is returned in "wordapp." The IDispatch interface is returned with a reference count of 1.
2. The code checks to ensure that an IDispatch pointer was returned. If not, it displays an error message and exits. The programmer can examine the status variable for the specific status return code.
3. The code calls Word.Basic methods to show the Microsoft Word window, open the document, replace the string, and save the modified document.
4. COMReleaseObject releases the single reference to the object's IDispatch interface so that Microsoft Word can terminate.

COM Objects The Microsoft PowerPoint version 7.0 type library contains a description of a number of COM objects and interfaces that make up the programmable interface to the Microsoft PowerPoint application. Figures 7 and 8 contain code generated by the Fortran Module Wizard from the Microsoft PowerPoint version 7.0 type library. Unlike Microsoft Word, which provides a single object that presents all of Word's programmable functionality, PowerPoint provides a hierarchy of objects. The top-level object, Application, is identified by the ProgID PowerPoint.Application.7. The Application object contains member functions that return a pointer to subordinate objects, including the Presentations

```
 ! Create a Word object and make it visible
1-  CALL COMCREATEOBJECT ("Word.Basic," wordapp, status)
2-  IF (wordapp == 0) THEN
        WRITE (*,
           '(" Unable to create Microsoft Word object; Aborting")')
        CALL EXIT(-1)
    END IF
3-  CALL Word_Basic_AppShow(wordapp, "," $STATUS=status)

    ! Open the document
    CALL Word_Basic_FileOpen(wordapp, filename, $STATUS=status)

    ! Replace all occurrences of the string
    CALL Word_Basic_EditReplace(wordapp, findstring, replacestring,
                    ReplaceAll=.TRUE., $STATUS=status)

    ! Save the file
    CALL Word_Basic_FileSaveAs(wordapp, filename, $STATUS=status)

    ! Release the Word.Basic object since we are done
4-  status = COMRELEASEOBJECT(wordapp)
```

Figure 6
Code from a User-written Fortran Program That Invokes Microsoft Word

object. The Presentations object consists of a collection of Presentation objects. A Presentation contains a member function that returns a pointer to its SlideShow object, and so on. By navigating this hierarchy, the developer can select a pointer to a particular object's interface. A code example in which we use some of the PowerPoint objects and interfaces to run a slide presentation from PowerPoint is given later in this section.

Figure 7 contains the interface description of the Presentations object's member function named Open. It is representative of the interfaces generated for all COM member functions. The procedure naming convention is *objectname_memberfunctionname*. The Open function opens an existing PowerPoint presentation.

1. The first argument to the procedure is always $OBJECT. It is a pointer to the object's interface. The remaining argument names are determined from the type information.
2. A BSTR is a length-prefixed string data type primarily for use by Automation objects. The wrappers generated for COM member functions convert from Fortran strings to BSTRs and vice versa.
3. A VARIANT is a data structure that can contain any type of Automation data. It contains a field that identifies the type of data and a union that holds the data value. The use of a VARIANT argument allows the caller to use any data type that can be converted into the data type expected by the member function.

```
    INTERFACE
1-    INTEGER*4 FUNCTION Presentations_Open($OBJECT, fileName,
                ReadOnly, Untitled, WithWindow, Open)
       USE DFCOMTY
       INTEGER*4, INTENT(IN)          :: $OBJECT    ! Object Pointer
       !DEC$ ATTRIBUTES VALUE   :: $OBJECT
2-     INTEGER*4, INTENT(IN)          :: fileName   ! BSTR
       !DEC$ ATTRIBUTES VALUE   :: fileName
3-     TYPE (VARIANT), INTENT(IN), :: ReadOnly ! (Optional Arg)
       !DEC$ ATTRIBUTES VALUE   :: ReadOnly
       TYPE (VARIANT), INTENT(IN), :: Untitled ! (Optional Arg)
       !DEC$ ATTRIBUTES VALUE   :: Untitled
       TYPE (VARIANT), INTENT(IN), :: WithWindow ! (Optional Arg)
       !DEC$ ATTRIBUTES VALUE   :: WithWindow
4-     INTEGER*4, INTENT(OUT)   :: Open
       !DEC$ ATTRIBUTES REFERENCE     :: Open
     !DEC$ ATTRIBUTES STDCALL   :: Presentations_Open
     END FUNCTION Presentations_Open
    END INTERFACE
5- POINTER(Presentations_Open_PTR, Presentations_Open)
```

Figure 7
Code Generated by Fortran Module Wizard from Microsoft PowerPoint, Interface Description of Open Function

4. Nearly every COM member function returns a status of type HRESULT. Therefore if a COM member function produces output, it uses output arguments to return the values. In this example, the Open argument returns a pointer to a PowerPoint Presentation object.
5. The interface of a COM member function looks similar to the interface for a DLL function with one major exception. Unlike a DLL function, the address of a COM member function is never known at program link time. To compute the address of a particular member function, the developer must get a pointer to an object's interface at run time. We have extended the DIGITAL Fortran 90 language to support a Pointer To procedure. Figure 8 shows an example of its use.

Figure 8 contains the wrapper generated by the Fortran Module Wizard for the Open function. The name of a wrapper is the same as the name of the corresponding member function, prefixed with a $. The numbers inserted at the left margin of the code example correspond to the following list.

1. The wrapper takes the same argument names as the member function interface.
2. Member function arguments of type BSTR are of type CHARACTER*(*) in the wrapper.
3. The wrapper computes the address of the member function from the interface pointer and an offset found in the interface's type information. In implementation terms, the sequence is the following: an interface pointer to a pointer to an array of function pointers called an Interface Function Table (see Figure 9).
4. The wrapper declares a local variable to hold the BSTR to be passed to the member function. The next line does the conversion.
5. Optional VARIANT arguments of a COM member function are represented by a VARIANT with distinguished values. OPTIONAL_VARIANT is defined in the DFCOMTY module with the distinguished values.
6. The offset of the Open member function is 60. The code assigns the computed address to the function pointer Presentations_Open_PTR, which was declared in Figure 7, and then calls the function.

```
1-  INTEGER*4 FUNCTION $Presentations_Open($OBJECT, fileName,
              ReadOnly, Untitled, WithWindow, Open)
      !DEC$ ATTRIBUTES DLLEXPORT   :: $Presentations_Open
      IMPLICIT NONE
      INTEGER*4, INTENT(IN)        :: $OBJECT   ! Object Pointer
      !DEC$ ATTRIBUTES VALUE       :: $OBJECT
2-    CHARACTER*(*), INTENT(IN)    :: fileName  ! BSTR
      !DEC$ ATTRIBUTES REFERENCE   :: fileName
      TYPE (VARIANT), INTENT(IN), OPTIONAL :: ReadOnly
      !DEC$ ATTRIBUTES REFERENCE   :: ReadOnly
      TYPE (VARIANT), INTENT(IN), OPTIONAL :: Untitled
      !DEC$ ATTRIBUTES REFERENCE   :: Untitled
      TYPE (VARIANT), INTENT(IN), OPTIONAL :: WithWindow
      !DEC$ ATTRIBUTES REFERENCE   :: WithWindow
      INTEGER*4, INTENT(OUT)       :: Open     ! IDispatch
      !DEC$ ATTRIBUTES REFERENCE   :: Open
      INTEGER*4 $RETURN
3-    INTEGER*4 $VTBL              ! Interface Function Table
      POINTER($VPTR, $VTBL)
      TYPE (VARIANT), :: $ VAR_ReadOnly
      TYPE (VARIANT), :: $ VAR_Untitled
      TYPE (VARIANT), :: $ VAR_WithWindow
4-    INTEGER*4 $BSTR_fileName     ! BSTR
      $BSTR_fileName = ConvertStringToBSTR(fileName)
5-    IF (PRESENT (ReadOnly)) THEN
        $VAR_ReadOnly = ReadOnly
      ELSE
        $VAR_ReadOnly = OPTIONAL_VARIANT
      Presentations_Open_PTR = $VTBL
      END IF
      ...
6-    $VPTR = $OBJECT              ! Interface Function Table
      $VPTR = $VTBL + 60           ! Add routine table offset
      Presentations_Open_PTR = $VTBL
      $RETURN = Presentations_Open($OBJECT, $BSTR_fileName,
              ReadOnly, Untitled, WithWindow, Open)
      $Presentations_Open = $RETURN
    END FUNCTION $Presentations_Open
```

Figure 8
Code Generated by Fortran Module Wizard from Microsoft PowerPoint, Wrapper for Open Function

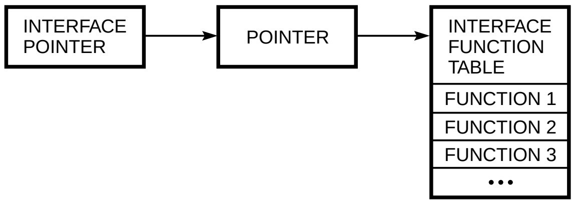


Figure 9
Interface Pointer to an Array of Function Pointers

In fact, PowerPoint provides dual interfaces. A dual interface is a combination of an IDispatch interface and COM member functions. The IDispatch interface of the dual interface can be used by Automation clients, and the COM member functions can be used by COM clients. This means that for PowerPoint, and any server that provides dual interfaces, the Fortran developer can choose to generate a Fortran module for the Automation interfaces or the COM interfaces. The Fortran interfaces generated by the Wizard likely will not be much different. COM interfaces typically provide better performance since there is less overhead in invoking COM member functions than dispinterface methods through the IDispatch Invoke member function.

Figure 10 shows code from a user-written Fortran program that invokes PowerPoint to run a slide presentation. The code example is annotated with numbers that correspond to the following list.

1. COMCLSIDFromPROGID and COMCreateObject request COM to create an object with the ProgID PowerPoint.Application.7, and to return a pointer to the object's IApplication interface.
2. The code gets the AppWindow object from the Application object and calls its Visible member function to make PowerPoint visible.
3. The code gets the Presentations object from the Application object and calls its Open member function to open a Presentation. Note that three of the arguments to Open are of the VARIANT data type. The code sets them to the values true and false.
4. The code gets the SlideShow object from the Presentation object and calls its Run member function to run the slide show.

DLLs When the Fortran Module Wizard reads the type information describing a DLL, it generates an interface description for each function in the DLL. It also generates Fortran-derived types for data structures defined in the DLL type information. This relieves the Fortran developer from manually translating header file descriptions to Fortran descriptions. The Wizard also provides the option of generating wrappers that convert from the Fortran representation of strings to the C representation of strings and vice versa. This option can be selected from the Wizard's initial dialog box (see Figure 2).

```
    ! Create a PowerPoint Application object
    ! and make the AppWindow visible
1-  CALL COMCLSIDFROMPROGID ("PowerPoint.Application.7,"
                             clsid, status)
    CALL COMCREATEOBJECT (clsid, CLSCTX_SERVER, IID_Application,
                             ppApplication, status)
    IF (ppApplication == 0) THEN
      WRITE (*, '(" Unable to create PowerPoint object; Aborting")')
      CALL EXIT(-1)
    END IF
2-  status = $Application_GetAppWindow(ppApplication, ppAppWindow)
    status = $ApplicationWindow_SetVisible(ppAppWindow, 1)

    ! Open the specified presentation
3-  status = $Application_GetPresentations(ppApplication,
                                            ppPresentations)
    vTrue%VT = VT_BOOL
    vTrue%VU%BOOL_VAL = VARIANT_BOOL_TRUE
    vFalse%VT = VT_BOOL
    vFalse%VU%BOOL_VAL = VARIANT_BOOL_FALSE
    status = $Presentations_Open(ppPresentations, filename,
                  vTrue, vFalse, vTrue, ppPresentation)

    ! Run the slide show
4-  status = $Presentation_GetSlideShow(ppPresentation, ppSlideShow)
    status = $SlideShow_Run(ppSlideShow, 1, ppRun)
```

Figure 10
Fortran Program to Invoke PowerPoint to Run Slide Presentation

Comparison of the Wizard to the Capabilities of Other Languages

Visual C++ version 5.0, Visual J++ version 1.1, and Visual Basic version 5.0 all have wizards that can read a type library and allow applications to use COM and/or Automation objects.

The Visual C++ ClassWizard can read a type library and create a class with all the functions of the IDispatch interface described in the library. Visual C++ version 5.0 also adds a preprocessor directive, #import. The #import directive reads a type library and generates two header files that contain the definitions of the COM objects defined in the type library.[13]

The Java Type Library Wizard within Visual J++ invokes the JavaTLB utility to convert the information in a type library into Java .class files. A Java .class file is the binary form of a Java class or interface.[14]

To use an object defined in a type library from Visual Basic, the developer must add a reference to the object using the Project menu, References command. The References dialog box allows the user to select from the list of registered type libraries in a manner similar to the Fortran Module Wizard.[15]

The Fortran Module Wizard is unique in the following ways. The Fortran 90 programming language does not inherently support objects. The Fortran Module Wizard employs a combination of language and run-time support to provide this capability. The supporting language features are modules and procedure pointers. The supporting run-time modules are DFCOMTY, DFCOM, and DFAUTO. The Fortran Module Wizard provides support for type libraries containing the descriptions of DLL routines.

Fortran Module Wizard Architecture

The architecture of the Fortran Module Wizard is fairly simple. The shell of the Wizard was generated by the Custom AppWizard within Visual C++. The inner workings of the Wizard consist of three major pieces:

- Type information reader
- Type symbol table
- Fortran code generator

Figure 11 shows a high-level data flow of the Fortran Module Wizard. The type information reader traverses the data structures in the type information and creates the type symbol table. The Win32 SDK provides a sample application named BROWSE OLE sample that is an example of traversing the information in a type library. The type symbol table is a symbol table similar to those used by compilers. It maps type names to the descriptions of types. For simplicity, the information is stored using the same data structures used by the type information. The Fortran code generator traverses the symbol table and generates a Fortran module.

The use of a symbol table allows for a complete separation of the functionality of the type information reader from the Fortran code generator. A code generator for another programming language could be easily substituted, as could another source of type information (for example, a C header file).

Future Directions

There are a number of possibilities for future work that would add to the capabilities provided by the Fortran Module Wizard.

- Fortran support for ActiveX controls. An ActiveX control is an Automation object. It is a reusable component that normally provides a user interface and is used in dialog boxes and other windows. The Fortran Module Wizard can generate a module that would allow a Fortran developer to use the methods and properties of an ActiveX control. However, additional functionality would be needed in the Fortran run-time libraries to make controls usable from a Fortran application. A control has to be placed in a special type of window called a Control Container. The Fortran run-time libraries do not currently contain support for a Control Container. In addition to methods and properties, a control can define events. An event allows a control to notify its container when something of interest happens to the control. For example, a "Button control" could define a "Clicked event."
- Fortran Windows Application Wizard. This Wizard could generate starter files for a Fortran Windows application. This would be especially useful if we were to implement the Fortran support for ActiveX controls.

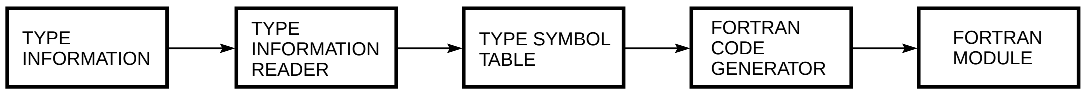


Figure 11
Data Flow of the Fortran Module Wizard

- Fortran modules from C header files. By replacing the type information reader described in the previous section with a C parser, we could generate Fortran modules directly from .h files. This would expand the set of services that are easily available to Fortran developers.
- Fortran Server Wizard. This Wizard would take a Fortran module provided by a Fortran developer and package it as a COM object. It would also generate a type library that describes the object. This object could then be used by any COM client, for example, Visual Basic, Visual C++, and Visual J++ applications.

References and Notes

1. *Digital Fortran Books Online* (Maynard, Mass.: Digital Equipment Corporation, 1997).
2. *Digital Fortran 90 Language Reference Manual* (Maynard, Mass.: Digital Equipment Corporation, 1997).
3. For a period of time, Microsoft used the name OLE to encompass all of its component integration technology, including COM. Now OLE is applied only to compound document technology.
4. K. Brockschmidt, *Inside OLE*, Second Edition (Redmond, Wash.: Microsoft Press, 1995).
5. K. Brockschmidt, "How OLE and COM Solve the Problems of Component Software Design," *Microsoft Systems Journal*, vol. 11, no. 5 (May 1996): 63–80.
6. D. Chappell, *Understanding ActiveX and OLE* (Redmond, Wash.: Microsoft Press, 1996).
7. *OLE 2 Programmer's Reference, Volume Two* (Redmond, Wash.: Microsoft Press, 1994).
8. *The Component Object Model Specification 0.9* (Redmond, Wash.: Microsoft Corporation, 1995).
9. Automation was originally called OLE Automation.
10. Before IDL and MIDL, Microsoft provided the Object Description Language (ODL) and a compiler named MKTYPLIB.
11. *Developer Studio Environment User's Guide* (Redmond, Wash.: Microsoft Corporation, 1997).
12. Microsoft Office 97 includes a new Office object model that offers another set of interfaces to Word services.
13. G. Shepherd, "Visual C++ Simplifies the Process for Developing and Using COM Objects," *Microsoft Systems Journal*, vol. 12, no. 5 (May 1997): 37–48.
14. G. Eddon and H. Eddon, "Understanding the Java/COM Integration Model," *Microsoft Interactive Developer*, vol. 2, no. 4 (April 1997): 56–68.
15. *Microsoft Visual Basic 5.0 Books Online* (Redmond, Wash.: Microsoft Corporation, 1997).

Biography

Leo P. Treggiari
Leo Treggiari is a consulting software engineer in the Core Technology Group. He was responsible for developing the Module Wizard in the DIGITAL Visual Fortran product for the Fortran programmer working in a Microsoft Windows environment. Previous to this work, he was project leader for the development of several programming tools, including the Motif toolkit. Leo came to DIGITAL in 1979 from Wang Laboratories. He holds a B.S. (1975, summa cum laude) in chemistry from Boston College and is a member of ACM.