VBA Solutions

MFC Programmer's Guide

MFC Programmer's Guide: Getting Started

Featuring Microsoft® Visual Basic® for Applications

Microsoft®

MFC PROGRAMMERS GUIDE

This guide is intended to acquaint you with the steps required to integrate Microsoft® Visual Basic® for Applications version 6.0 into host applications running on Microsoft Windows® 32-bit operating systems. It features the Application Programmability Component (APC), which simplifies Visual Basic for Applications integration. See Introducing APC for more information.

Prerequisites

To use APC effectively in MFC applications, you should be familiar with the following:

- Component Object Model (COM), which is the foundation of Automation (formerly OLE Automation)
- Automation
- The Visual C++ programming language
- The Microsoft Foundation Class (MFC) Library

To get the most out of this guide, you should be familiar with the following:

- The Visual C++ programming language
- C++ programming using the Active Template Library (ATL)
- The Microsoft Windows 95 or Windows NT® programming environment
- COM
- Programming Automation servers

PREPARING THE DEVELOPMENT ENVIRONMENT FOR INTEGRATION

To build an application enabled for Visual Basic for Applications using the Visual C++ development environment, you will need to do the following:

- Install Microsoft Visual C++ version 5 (Service Pack 3 required) or later
- Install the Visual Basic for Applications 6.0 Software Development Kit
- Ensure that your list of directories include files contains the following (assuming that C:\vba6sdk is your SDK install directory):

```
C:\vba6sdk\include
C:\Program Files\Common Files\Microsoft Shared\vba\vba6
C:\Program Files\Microsoft Office\Office
```

If you installed the debug version of Visual Basic for Applications 6.0, use the following directories:

```
C:\vba6sdk\include
C:\Program Files\Common Files\Microsoft Shared
Debug\vba\vba6
C:\Program Files\Microsoft Office Debug\Office
```

- Ensure that your list of directories for library files contains the Visual Basic for Applications library directory:

```
C:\vba6sdk\lib
```

Note: *To build applications from the command line, you must add these directories to the INCLUDE and LIB paths in your command-line environment.*

INTRODUCING APC

This section briefly describes the Microsoft Application Programmability Component (APC), which is a COM object that simplifies integration of Visual Basic for Applications. This section also steps through the key tasks involved in using APC to integrate Visual Basic for Applications 6.0 into an MFC application. APC exposes a number of Application Programming Interfaces (APIs) on top of the core Visual Basic for Applications API to handle tasks formerly done by the developer.

APC is recommended for integrating Visual Basic for Applications. Most development environments that are capable of creating, calling, and handling events from COM objects can integrate Visual Basic for Applications.

To integrate APC into an MFC application, you need to provide minimal support for ATL, expose your application to APC using Automation interfaces, and override a few MFC methods and message handlers.

APC includes the following items and functionality:

- **Automation interfaces** that allow integration of Visual Basic for Applications into applications written in Visual Basic.
- **Template-based classes** that ease integration of Visual Basic for Applications into MFC and ATL applications. See "Template-Based Classes," later in this topic, for more information.
- **Default host interfaces** that eliminate much of the work needed to integrate Visual Basic for Applications into hosts written in C or C++. These interfaces handle tasks such as instantiating Visual Basic for Applications, managing project items, hosting controls, storing Visual Basic for Applications projects, recording macros, and enabling advanced features such as host classes and digital signing. See Getting Visual Basic for Applications into your MFC Project.
- **Message loop integration** for all popular development platforms, such as Visual Basic, C++, and MFC.
- **An APC Reference** documenting each APC interface. To view the APC Reference, go to the Contents section of the Visual Basic for Applications SDK Help file (vba6sdk.chm).

Template-Based Classes

APC includes a set of template-based classes, defined in the SDK header file ApcMfc.h, which make it easy to integrate Visual Basic for Applications into an MFC application. These classes provide a number of base classes that support Visual Basic for Applications MFC-style persistence, MFC-style message loop handling, and MFC-style OLE control containment. By overriding the default implementations in these base classes, you can customize your integration with little effort. The SDK header file ApcCpp.h provides support for non-MFC application, document, project item, and control containment classes.

For details on the contents of APC, see the APC Reference in the Visual Basic for Applications SDK Help file (vba6sdk.chm).

Using APC

This guide is intended to show you how to:

- Display and hide the Visual Basic for Applications Integrated Development Environment (IDE)
- Create Visual Basic for Applications projects and associate them with your application’s documents
- Create, load, and store Visual Basic for Applications projects
- Expose your application’s object model to Visual Basic for Applications end users

The example code here is based on Step 1 of the VBACalc sample program, which integrates Visual Basic for Applications into a calculator program similar to the one that comes with Windows. You can find the source code for VBACalc in the `\Samples\VBACalc` directory of your distribution.

This is not intended to be a complete programmer’s guide to integrating Visual Basic for Applications. Some of the techniques outlined here and used in VBACalc have been chosen for simplicity, rather than completeness.

To add Visual Basic for Applications support to your own MFC project, you must do, at a minimum the following:

- Integrate Visual Basic for Applications into your host application to the point where it can display the IDE and run programs. See Getting Visual Basic for Applications into Your MFC Project.
- Add storage support to Visual Basic for Applications so it can load and store projects. See Adding Project Support.

GETTING VISUAL BASIC FOR APPLICATIONS INTO YOUR MFC PROJECT

This section discusses the two main steps you need to integrate Visual Basic for Applications into your own MFC program. This steps are adding application-level support and adding project support.

An MFC program has an application class normally derived from the standard MFC class **CWinApp**. A Visual Basic for Applications-enabled MFC program using APC should instead derive from the template class **CApcApplication**, to which you pass in a base application class (usually **CWinApp**) through the **AppBase** template parameter. The term host application class refers to such a **CApcApplication**-derived template class. This is to emphasize that APC is managed through its data member **ApcHost**, which connects the **AppBase** application member to APC.

The host application class also manages the lifetime of the global application object, which is an Automation server that appears as part of the project in Visual Basic for Applications code. Its functions can be made available globally, without qualification and without requiring that the Visual Basic for Applications user instantiate the global object.

Other tasks at the application level include managing window modes and accelerators, displaying the IDE when users request, and making sure Basic code has been halted before shutting down the IDE.

Adding project support is straightforward in an MFC application, because the MFC template-based classes that come with APC contain a **CApcDocument** template class. The **CApcDocument** base class provides the overrides and implementations necessary to create, register, load, and save projects using OLE storage if you pass in an MFC document class with compound document support.

To add application-level support:

- Use **CApcApplication** as the application class for your host. See Using the APC Application Class.
- Use the **CApcHost::ApcHost.Create** data member of your host application class to initialize Visual Basic for Applications and pass in a pointer to the global application class you will use for the host object model. See Initializing Visual Basic for Applications.
- Ensure that your host manages its accelerators and window modes properly by forwarding WM_ACTIVATE and WM_ENABLE messages to APC. See Coordinating Window Management with the Visual Basic for Applications IDE.
- Display the IDE when appropriate. See Displaying the Integrated Development Environment.
- When the user shuts down Visual Basic for Applications, ensure that no Basic code is executing, unwind the message loop stack, and give the user a chance to save any unsaved standalone projects. This means writing an **OnClose** handler for the main frame window and calling **IApc::CanTerminate** to check for unsaved standalone projects. See Terminating Visual Basic for Applications.
- Shut down Visual Basic for Applications and clean up as appropriate. Override the standard MFC method **CWinApp::ExitInstance** to terminate Visual Basic for Applications, and provide your own cleanup code. See Terminating Visual Basic for Applications.

To add project support:

- To find out more about adding support for projects and OLE storage, see Adding Project Support.

USING THE APC APPLICATION CLASS

To gain access to the root APC object, you should derive your MFC Application class from the **CApcApplication** template class.

MFC applications use a global instance of a class derived from **CWinApp** to manage the main message loop, coordinate the creation of frame windows, and create documents from document templates. Your APC project should use the **CApcApplication** template class for this purpose. Doing so will provide you with a member variable, **ApcHost**, within the class—a smart pointer to the root APC object. Because **CApcApplication** derives in part from **CWinApp**, all of your MFC application services are still available.

Your application class should be declared using **CApcApplication** instead of **CWinApp**. It should also bring the APC namespace into scope for the header file. **CApcApplication** takes two template parameters. The first parameter is the name of your MFC application class; the second is the name of the class that **CApcApplication**
should be derived from (i.e. your previous base class). The second parameter is optional and defaults to **CWinApp**.

To change your CWinApp definition to CApcApplication:

- Open the header file containing the declaration application class, which in most MFC projects is a class deriving from **CWinApp**.
- Add an MSAPC namespace declaration to the header file of the application class, as in the following example: using namespace MSAPC;
- Change the application class to derive from the CApcApplication template class, as in the following example:
  class CVBACalcApp : public CApcApplication<CVBACalcApp>

INITIALIZING VISUAL BASIC FOR APPLICATIONS

The host application class is responsible for initializing Visual Basic for Applications through the **CApcHost::ApcHost.Create** member function. It is best to defer initialization until Visual Basic for Applications is needed.

The minimum parameters you should pass to **ApcHost.Create** include a parent window handle, text for window captions (or NULL to use the caption of the parent window), an **IDispatch** pointer to a global automation object, and a Visual Basic for Applications license key. The example that follows passes in a temporary evaluation license key. In the application you ship, it is necessary to pass in a valid license key, which you receive after executing a license agreement with Microsoft. If you use the evaluation license key in your shipping application, **Create** fails after the expiration date.

To begin the Visual Basic for Applications session using APC:

The following example from **CVBACalcApp::CreateAPCHost** illustrates the use of **CApcHost::ApcHost.Create**.

```
HRESULT hr = NOERROR;
 // Create Visual Basic for Applications. APC will call ApcHost.Destroy() on
application exit.
CString strAppName, strLicKey;
strAppName = m_pszAppName;

strLicKey =
"04054348435D545D5464B4ADD5EC32115371BE988D9CC4E3C413"

DWORD dwLCID = MAKELCID(m_nVBALanguage, SORT_DEFAULT);

hr = ApcHost.Create(GetMainWnd()->m_hWnd,
strAppName.AllocSysString(),
    GetIApplication(FALSE),
    strLicKey.AllocSysString(),
    dwLCID);

if(FAILED(hr))
{
    AfxMessageBox("Error initializing Visual Basic for Applications.");
    return hr;
}
return hr;
```

COORDINATING WINDOW MANAGEMENT WITH THE VISUAL BASIC FOR APPLICATIONS IDE

Visual Basic for Applications tracks the active component so it can coordinate accelerators and manage other window coordination, such as the modality of UserForms. To manage the active component, you need to forward the WM_ACTIVATE and WM_ENABLE messages to APC. The following code provides the proper implementations of these event handlers:

```
void CMainFrame::OnEnable(BOOL bEnable)
{
        CFrameWnd::OnEnable(bEnable);
        GetApp()->ApcHost.WmEnable(bEnable);
}
void CMainFrame::OnActivate(UINT nState,CWnd* pWndOther,BOOL
bMinimized)
{
CFrameWnd::OnActivate(nState, pWndOther, bMinimized);
GetApp()->ApcHost.WmActivate(nState);
}
```

```
        void CVBACalcApp::OnVbaIde()
        {
        if(ApcHost)
        CApcApplication<CVBACalcApp>::OnVbaIde();
        }
```

DISPLAYING THE INTEGRATED DEVELOPMENT ENVIRONMENT

A single member function allows you to display the Visual Basic for Applications integrated development environment (IDE).

To display the IDE:

- Call **CApcApplication::OnVbaIde** as follows:

TERMINATING VISUAL BASIC FOR APPLICATIONS

Quitting Visual Basic for Applications requires you to ensure that no message loops are on the stack, that no end user Basic code is running, and that no standalone projects need to be saved before quitting.

To ensure that no message loops are on the stack, use **CApcHost::ApcHost.WmClose**, which notifies APC of a WM_CLOSE event on the main window and lets you query the state of Visual Basic for Applications execution. The fTerminated parameter of **CApcHost::ApcHost.WmClose** returns **TRUE** if it is safe to terminate running code, and **FALSE** if Visual Basic for Applications is still unwinding the stack. Continue posting **WM_CLOSE** messages in the **FALSE** case.

CApcApplication overrides **CWinApp::ExitInstance** to destroy Visual Basic for Applications, close down Visual Basic for Applications in an orderly fashion, and allow you to perform any cleanup necessary during application shutdown. If you override **ExitInstance**, make sure you call this base class version from your override.

Any copy of Visual Basic for Applications can be used for standalone projects, whether or not the version you licensed supports them. Users can upgrade Visual Basic for Applications at any time to Microsoft Office Developer, which supports the creation of standalone projects. Before exiting, the host should therefore call **IApc::CanTerminate**, which checks each standalone project and saves if necessary. In the event of a save, a dialog box asks if the user wishes to save the project. If **IApc::CanTerminate** returns **VARIANT_TRUE**, either no standalone projects existed or the user didn’t wish to save them, and the host may continue exiting.

To close down Visual Basic for Applications:

- Write an OnClose handler for the application's top-level frame window. OnClose is the standard MFC member function called when an application is about to terminate.
- Call **IApc::CanTerminate** to determine whether the user wishes to save any unsaved code in standalone projects. This code is called from **CMainFrame::OnClose** in the following VBACalc example:

```
BOOL  bTerminated;
GetApp()->ApcHost.WmClose(bTerminated);
if(!bTerminated) {
// unwind stack and try again
PostMessage(WM_CLOSE, 0, 0);
return;
}
// Prompt the user to save changes

if(!!GetApp()->ApcHost)
{
VARIANT_BOOL bCanTerminate;
GetApp()->ApcHost->APC_RAW(CanTerminate)(&bCanTerminate);
if (bCanTerminate == VARIANT_FALSE)
return;
}
CFrameWnd::OnClose();
```

- If you override **CWinApp::ExitInstance** in your application, be sure to call the proper base class version of it from your override:

```
 int ret = CApcApplication<CVBACalcApp>::ExitInstance();
 ...
return ret;
```

To make sure no Visual Basic for Applications end user code is running:

- To halt execution of the end user's Visual Basic for Applications project, call **IApc::End**. A good place to do this is in your implementation of the standard **MFC CDocument::DeleteContents** method, which is called when you delete the data in your document. The following example from VBACalc's CalcDoc.cpp file illustrates this:

```
if(!!((CVBACalcApp*)AfxGetApp())->ApcHost)
{
((CVBACalcApp*)AfxGetApp())->ApcHost->APC_RAW(End)(NULL);
...
```

THE HOST OBJECT MODEL

Your application may provide an object model to the Visual Basic for Applications end user. The root object of its hierarchy is called a *global application class*, which is an Automation server that can be built in any manner supporting **IDispatch**. The name of the root Automation object in these examples, and in most commercial products that use Visual Basic for Applications, is **Application**. The **Application** object is normally tagged as an appobject in the type library, meaning that the Visual Basic for Applications end user does not need to declare it or even precede its method invocations with the CoClass name, **Application**. For example, if the **Application** object has a method named **Version** that returns a version number, it can be invoked from end user code as follows:

```
Debug.Print Version
```

The instance of the global application class that is connected to the host at run time is called the *global application object*.

The suggested technique for writing your global application class is to use the **ApcDual** template classes for the creation of this object in an MFC application. Doing this keeps the COM interfaces and MFC implementation in separate or "peer" classes.

An alternative to using **Application** is to use MFC’s **CCmdTarget**. However, the default implementation of such objects in MFC does not supply the type of information needed by Visual Basic for Applications, and will need modification to work properly. Further, using the **ApcDual** template classes provides dual interface support for your COM objects—a significant performance benefit.

The following steps are required to add a global automation object to your host application:

- Declare a global application class and its associated COM interface class using the **ApcDual** interface classes. The object model class is derived from the standard MFC **CCmdTarget** Automation class, and is passed into the **CApcDualAgg** template class. See Creating the Global Application Class.
- Inherit the MFC class from either the **CApcDualAgg** or the **CApcDualDyn** template class. Pass the inherited class as a template argument to the **ApcDual** class.
- Declare the COM class using ATL, or use the ATL Object Wizard.
- Implement **IApcDual** in the ATL class by inheriting from **IApcDualImpl**.
- Implement **IDispatch** in the ATL class.
- Implement **IProvideClassInfo** in the ATL class.
- Connect the global automation object by passing its **IDispatch** to APC at creation time. See Connecting the Global Application Object.

CREATING THE GLOBAL APPLICATION CLASS

The host object model can be a **CCmdTarget**-derived Automation server that uses the **ApcDual** template classes to provide the COM interface in a separate class, as in the VbaCalc sample. (It can also be a straight ATL class or any other implementation of the interface.) By convention, the name of this CoClass, as it appears to the Visual Basic for Applications programmer, is **Application**. The internal names of the C++ classes in the example code reflect this.

In this example, the matching **ApcDual** class used to expose member functions of class **CApplication** as Automation servers is **CAtlApplication**. See “Using the APC Dual Template Classes” for more information on how these classes interact.

To declare the object model class:

- Derive your **CApplication** class from the MFC **CCmdTarget** class, its matching **CAtlApplication** class, and the **IDispatch**-derived **IDualApplication** class, as in the following example:

```
class CApplication : public CApcDualAgg<CCmdTarget, CAtlApplication,
IDualApplication, &IID_IDualApplication>
{
...
```

CONNECTING THE GLOBAL APPLICATION OBJECT

When Visual Basic for Applications is initialized via **CApcHost::ApcHost.Create**, you are given a chance to pass in a global application class. In the VBACalc example, the host application class has a member function, **GetIApplication**, which returns the **IDispatch** for the global application class.

To connect the global application object to the host application class at initialization time:

- Pass the global application object in to the *pDisp* parameter (the third argument in the example below) to **CApcHost::ApcHost.Create** as follows:

```
hr = ApcHost.CreateGetMainWnd()->m_hWnd,
strAppName.AllocSysString(),
GetIApplication(FALSE),
NULL,
dwLCID);
```

For more information on initialization, see Initializing Visual Basic for Applications.

ADDING PROJECT SUPPORT

A Visual Basic for Applications project is a collection of project items, their interactive behavior, and their code. Project items are the building blocks of a project. They include forms, classes, modules, class modules, host-provided project items such as host modules (sometimes called document project items), and—new in Visual Basic for Applications 6.0—host classes with code behind them. The project controls how project items interact and communicate. The host provides storage and management for the project (if desired) and for the host's own persisted data. Throughout this Programmer's Guide, the term document by itself refers to the host's persisted data, which includes the Visual Basic for Applications project.

Visual Basic for Applications expects you to persist projects using COM's structured storage via **IPersistStorage**. Because a Visual Basic for Applications project can be saved into an MFC document which uses structured storage and a similar storage management scheme, APC provides a template class called **CApcDocument**. This makes project persistence convenient and familiar to MFC programmers. Using **CApcDocument** gives you project management and storage in a convenient, tested package.

Note: *Placing both your host's data and the project in the same file is one method of saving a project, but you are not required to store a project in the same file as your document. You don't even need to save it in physical storage; for example, you can provide project storage in memory for the current session only, freeing it when the user quits or a different project is loaded.*

You will need to take the following steps to add project support using CApcDocument:

- Create a document class for your project. The easiest way to do this is to use **CApcDocument**. See Creating a Document Class for Your Project.
- Provide storage for your project. This is part of the implementation of **CApcDocument**. See Creating Storage for Your Project.
- Allow the user to save the project at will by choosing **Save** from the **File** menu or clicking the **Save** button. See Handling Save Notifications.

CREATING A DOCUMENT CLASS FOR YOUR PROJECT

In an AppWizard-created MFC project with Automation support, the project's document class normally derives from **COleServerDoc**. The APC MFC framework, however, provides its own template class, **CApcDocument**, to create, register, load, and save projects.

Using **CApcDocument** means that the overrides of the MFC document methods, such as **OnNewDocument** that it provides, automatically create, register, load, and save the APC project and its contents. The second template parameter of **CApcDocument** derives from the MFC class you used. It defaults to **COleServerDoc**, so if you had previously derived from that, you can omit the second parameter, as in the example below. This inserts **CApcDocument** into the proper place in your hierarchy. You must derive at least from **COleDocument** to get MFC's compound document support.

To create a document class deriving from CApcDocument:

- Change your MFC document class to derive from the CApcDocument template document class, and pass your class name into it. In the example below, the class name is **CVBACalcDoc**, and **COleServerDoc** is the implied second template parameter.

```
class CVBACalcDoc : public CApcDocument<CVBACalcDoc>
```

ABOUT PROJECT STORAGE

Visual Basic for Applications requires an OLE storage object for a project when it starts up, but MFC applications typically defer the creation of a storage object until the document is saved. The APC MFC framework provides an implementation of the virtual **OnNewDocument** method.

The CApcDocument base class implementations of **OnNewDocument**, **SaveToStorage**, and **LoadFromStorage** handle the actual creation and persistence of the document to a storage object.

HANDLING SAVE NOTIFICATIONS

When an end user chooses **Save** from the **File** menu or clicks the **Save** button in the IDE, the **ApcProject_Save** event fires. You should handle this in your host in such a way that the end user can save at will.

To implement the ApcProject_Save event handler:

- Write a Save handler in your document class to persist the project on demand. The following code illustrates one way of doing this.

```
HRESULT CVBACalcDoc::ApcProject_Save()
{
OnFileSave();
return NOERROR;
}
```

HALTING VISUAL BASIC FOR APPLICATIONS WHEN DOCUMENT CONTENTS ARE DELETED

MFC calls its **CDocument::DeleteContents** member function just before a document is destroyed. For example, **DeleteContents** is called from **CDocument::OnCloseDocument** after the view has been closed and the document's data can be cleaned up, and just before **CDocument::OnNewDocument** to ensure that the document is empty. Before calling **DeleteContents**, the host should ensure that that Visual Basic for Applications execution has been halted by calling **IApc::End**.

To call the APC DeleteContents implementation:

- The following code demonstrates how VBACalc halts Visual Basic for Applications and calls the APC base class implementation of **DeleteContents**. In this example, the application class is **CVBACalcApp**.

```
void CVBACalcDoc::DeleteContents()
{
// Stop Visual Basic for Applications execution
if(!!((CVBACalcApp*)AfxGetApp())->ApcHost)
{
((CVBACalcApp*)AfxGetApp())->ApcHost->APC_RAW(End)(NULL);
}
CApcDocument<CVBACalcDoc>::DeleteContents();
}
```

WHERE TO GO FROM HERE

Once you are ready for more advanced Visual Basic for Applications features, you may have questions that go beyond the scope of this guide. We strongly recommend you contact your authorized Visual Basic for Applications agent, who can provide you with technical support on your integration and discuss various integration models. See Other Resources for Visual Basic for Applications in the Welcome Guide for authorized agent contact information.