Building Component Software With Visual C++ and the OLE Custom Control Developer's Kit

Click to open or copy the Smile project files

Eric Lang

Component software is a popular term these days. The idea that software can be packaged into small, powerful, self-contained units that can be put together to build apps is in many respects a realization of the object-oriented model for software development that was designed to address. The difference is that software components use a binary model, while typical object-oriented programming requires a source-code model. With component-based software development, the developer using the components didn't need to know how they were written, just the interfaces that they expose.

Some would argue that the first commercially successful model for component software was the Visual Basic Extension (VBX) model. The VBX architecture was developed as a means of extending the Visual Basic programming environment. It was intended that programmers would create new “custom controls” to augment the standard Windows™ system controls, such as push buttons, scrollbars, check boxes, and so on. It turned out that planned or not, the VBX architecture was sufficiently rich that more powerful software components were developed as VBXs. (OLE, multimedia viewing, modern communication, three-dimensional modeling add-ons have all been packaged as VBXs with simplified interfaces, “Custom control” turned out to be too narrow a term for the way VBXs were created and used.

As popular as it is, the VBX architecture was not originally designed as an open standard for component software. There are several reasons why this architecture does not scale well to other platforms; for instance, the VBX API and data structures are inherently 16-bit oriented. The APIs make extensive use of “based pointers,” which means that the APIs are tied closely to the 16-bit segmented Intel® architecture. (Based pointers are near pointers, only two bytes in size because they contain only the offset portion of an address. The segment portion of the address is stored separately and is combined with the offset when needed.) The APIs actually make calls through a specific address in the container application’s stack. The APIs were not designed but instead evolved out of a proprietary application, which is to say that they are not as clean and consistent as you would like to see in an open standard.

Even 16-bit applications that host the VBX API, such as Visual Basic and Visual C++,™ have a difficult time being consistent with each other. For example, Visual C++ version 1.5, supports only version 1 of VBX controls (VBX is currently at version 3). Contrast this to OLE 2, which was publicly reviewed and designed from the ground up to be a portable open standard. OLE 2 technology and the Component Object Model (COM) were maturing at about the same time that people were considering how to get VBX functionality to new platforms, and it was obvious that OLE 2 solved most of the portability and extendability problems associated with VBXs.

OLE 2 was originally used in fairly large and complex applications such as Word 6.0 and Microsoft Excel 5.0; even now some skeptics don’t believe that OLE 2 can be applied to smaller, faster components—a myth that OLE Custom Controls dispel. OLE Custom Controls utilize OLE 2 to achieve parity with VBXs in size and speed, and to surpass them in functionality.

In this article, which presumes you have a basic grasp of OLE 2 fundamentals, I’ll discuss what OLE Custom Controls are all about. Then I’ll explain the CDK. Some would argue that the first commercially successful model for container software was the Visual Basic Extension (VBX) model. The VBX architecture was developed as a means of extending the Visual Basic programming environment. It was intended that programmers would create new “custom controls” to augment the standard Windows™ system controls, such as push buttons, scrollbars, check boxes, and so on. It turned out that planned or not, the VBX architecture was sufficiently rich that more powerful software components were developed as VBXs. (OLE, multimedia viewing, modern communication, three-dimensional modeling add-ons have all been packaged as VBXs with simplified interfaces, “Custom control” turned out to be too narrow a term for the way VBXs were created and used.

As is the case with VBXs, there are three key interface types that define an OLE Custom Control: properties, events, and methods.

Properties are named values in the control; for instance, color or text. They can be state identifiers or behavioral characteristics such as whether or not a push button can be tabbed to. A property value can be dynamic, that is, computed only when requested. A property can also be an aggregate of elements, like an array, a font, or a picture. Parameterized properties, also called property arrays, are homogenous collections of values exposed as a single property of the control.

OLE Custom Controls classify properties as four different types: stock, ambient, extender, or custom. (Ambient and extender properties are new to OLE Custom Controls.) Stock properties are standard properties that most controls need to implement. To save programmers from having to reimplement the same code over and over and to ensure consistency, Microsoft provides implementations for these in the control run-time DLL that comes with the CDK (more on this later). A Caption is a good example of a stock property. All stock properties can also be overridden to provide a custom implementation.

Ambient properties are read-only properties that simply provide information about the container, such as ForeColor. This makes it possible for the control to synchronize its common properties with the container. Extender properties, such as Left and Height, are properties that are also managed by the container. Custom properties are those you implement yourself.

Events are triggered by the control in response to some action happening to the control; such as a mouse click or a key press. The control transforms these actions into event notifications that are sent to the container; the container can then execute code based on the event. Events can also have parameters. For instance, in addition to sending an event that indicates mouse movement, you may also send parameters indicating current x and y positions of the mouse. Like properties, events can be classified as either stock or custom.

Methods are functions within the control that can be called externally by the container. For example, C++ code in the container may make a call to a method within the control that causes the control to update its appearance. Methods can also be either stock or custom (new to OLE Custom Controls).

OLE Custom Controls
In OLE 2 lingo, an OLE Custom Control is a compound document object implemented as an in-process server DLL that supports OLE automation and visual editing as an inside-out object. More simply, it is a standard OLE 2 object that can be embedded within a container, in-place activated, and can utilize OLE automation. Being inside-out means that the object is usually active when visible and becomes UI-active on a single click rather than a double click. As an in-process server, an OLE Control is in the same process space as the container, so it can communicate with the container using fast function calls rather than slower Lightweight Remote Procedure Calls (LRPC).

Beyond all that, an OLE Custom Control must support the new control-specific interfaces. This requires new interfaces in both the container application (the EXE) and the control (the DLL). Figure 2 lists the interfaces required by an OLE Custom Control container and an OLE Custom Control. If you don't implement the control interfaces, you are left with a compound document container and a standard embeddable object implemented as an in-process server DLL.

Figure 2 OLE Container and Control Interfaces

General OLE Interfaces

<table>
<thead>
<tr>
<th>Container (EXE)</th>
<th>Control (DLL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I OleClientSite</td>
<td>I OleObject</td>
</tr>
<tr>
<td>I OleInPlaceFrame</td>
<td>I OleInPlaceObject</td>
</tr>
<tr>
<td>IAdviseSink</td>
<td>I OleInPlaceActiveObject</td>
</tr>
<tr>
<td></td>
<td>I DataObject</td>
</tr>
<tr>
<td></td>
<td>I ViewObject</td>
</tr>
<tr>
<td></td>
<td>IDispatch</td>
</tr>
<tr>
<td></td>
<td>IDispatch (Event)</td>
</tr>
<tr>
<td></td>
<td>IDispatch (Ambient)</td>
</tr>
<tr>
<td></td>
<td>IDispatch (Extender)</td>
</tr>
<tr>
<td></td>
<td>IPersistStreamInit</td>
</tr>
<tr>
<td></td>
<td>IPersistStorage</td>
</tr>
</tbody>
</table>

Specific Control Interfaces

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IContainerSite</td>
<td>IDispatch</td>
</tr>
<tr>
<td>[ISimpleFrameSite]</td>
<td>(Prop/Meth)</td>
</tr>
<tr>
<td>IDispatch (Ambient)</td>
<td>(Prop/Meth)</td>
</tr>
<tr>
<td>IDispatch (Extender)</td>
<td>ConnectionPointContainer</td>
</tr>
<tr>
<td>IDispatch (Event)</td>
<td>SpecifyPropertyPages</td>
</tr>
<tr>
<td>IBoundObjectSite</td>
<td>PropertyNotifySink</td>
</tr>
</tbody>
</table>

You can read about the standard OLE 2 interfaces in various OLE 2 texts, such as Kraig Brockschmidt's Inside OLE 2.0 (Microsoft Press). Figure 3 briefly describes the new interfaces required by OLE Custom Controls.

Figure 3 Interface Descriptions for OLE 2 Controls

Container Interfaces

I OleClientSite Used by the control to communicate with its client site. Specifically, it supports mnemonic keys, focus changes, and coordinate transformations.

I SimpleFrameSite (optional) Used by a simple frame control to allow its container to hook window messages during the execution of the control's window procedure.

IDispatch (Ambient) Used by the control to access the container's ambient properties.

IDispatch (Extender) Implemented by the container to "extend" the set of properties that a control appears to have. Specifically it adds the container's extender properties to the list of properties exposed by the control.

IDispatch (Event) The interface that a control calls to fire its events.

IBoundObjectSite Used by the control to send data binding notifications and additional information to the container about the state of a bound property. The additional information might, for instance, be that the user started to edit the value, but then canceled.

PropertyNotifySink Used by the control to report changes in its property values.

Control Interfaces

IDispatch (Properties/Methods) The control interface that exposes the control's properties and methods to OLE automation clients.

IPersistStreamInit An extended version of the I PersistStream interface. If a container uses this interface, it is obligated to call IPersistStreamInit::InInit if it does not call IPersistStreamInit::Load, IPersistStreamInit::LoadNew, or IPersistStreamInit::InNew.

I Control Used by the container to communicate with the control regarding mnemonic keys, changes in ambient properties, and freezing/unfreezing of events.

I ConnectionPointContainer Provides access to the control's connection points. A control typically has two connection points: one for its events, and one for its property notifications.

ISpecifyPropertyPages Enumerates the class IDs of the property pages that can be used to view and change the control's properties.

The OLE Custom Control Developer's Kit

If you had to build OLE Custom Controls armed only with a technical specification defining the OLE 2 interfaces in Figure 2, you would find yourself facing, ahem, a nontrivial task. Fortunately, the Microsoft OLE Custom Control Developer's Kit (CDK) will be released soon and will make the job of creating OLE Custom Controls much easier. The essential ingredients of the CDK are listed in Figure 4.

Figure 4 Preliminary List of CDK Components

- 16-bit Control Runtime DLL (OC25.DLL)
- 32-bit Control Runtime DLL (OC30.DLL)
- ControlWizard (generates code for controls that are both 16- and 32-bit)
- Enhanced ClassWizard for Visual C++ 1.5 (the ClassWizard in Visual C++ 2.0 comes "control aware")
- 16-bit Test Container (TSTCON16.EXE)
- 32-bit Test Container (TSTCON32.EXE)
- MFC Source Code for OLE Custom Controls (supports both 16- and 32-bit controls)
- Control Sample Source Code (can be built as either 16- or 32-bit controls)
- Tutorials
- Full On-line Documentation

When the CDK is combined with either Visual C++ version 1.5 or Visual C++ version 2.0 you have all the necessary pieces to make 16- or 32-bit OLE Custom Controls. At its heart, the CDK contains new C++ classes that extend MFC, enabling MFC to support the new OLE Custom Control standard. This functionality is packaged in the form of a run-time DLL for either 16- or 32-bit controls. As usual, the MFC source code is provided.

The CDK also provides a new Visual C++ wizard, the ControlWizard, which can create an initial OLE Custom Control similar to the way that the Visual C++ AppWizard creates an initial MFC Windows-based application. An enhanced version of the Visual C++ ClassWizard is also included to support Visual OLE Custom Control features such as event handling.

The Test Container is your general-purpose testing and debugging tool for OLE Custom Controls. It allows you to test many aspects of your
control as you develop it, such as inserting the control into a container, testing the control's toolbar bitmap, testing its properties, events, and methods, and testing its persistence. (You could also use Microsoft Access® 2.0 as a test container for your 16-bit OLE Custom Controls. It also allows you to test all the features just mentioned. Eventually other Windows-based applications, such as Visual Basic and FoxPro, will also support OLE Custom Controls.) Finally, a full set of online documentation, including tutorials, samples, reference manuals, and programming manuals is included in the CDK. It provides separate 16- and 32-bit components and libraries where necessary.

Figure 5 illustrates how a control built with the CDK works. The container application is an executable program (EXE) with one or more client sites. When you want to insert a new control into the container, the container creates a new client site to receive it. Each client site can handle all communication between the container and a control (OCX) embedded within the container. A control is essentially a DLL with an OCX extension.

![Figure 5 Architecture for a CDK Control](image)

OLE Custom Control communication is based on Component Object Model (COM) interfaces. A COM interface is usually represented in diagrams by a line with a circle on the end. COM interfaces are actually arrays of function pointers: you communicate with an object by calling a function within an interface. To use an interface, you get a pointer to a COM interface (represented by an arrow), offset from that pointer to index into the function you want, and then execute the function. As you see in Figure 5, the container's client site exposes interfaces to the control, the control exposes interfaces to the container's client site, the control exposes interfaces directly to the container, and the container exposes interfaces outside the client site. The run-time DLL performs several services for the control (see Figures 6 and 7): it implements the control interface based on MFC. It includes support for the stock properties, stock events, and stock methods, the stock property pages, and the property page frame.

![Figure 6 The Control Run-time DLL](image)

Figure 7 Stock Support in the Control Run-time DLL

<table>
<thead>
<tr>
<th>Stock Properties</th>
<th>Stock Events</th>
<th>Stock Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackColor</td>
<td>Click</td>
<td>AboutBox*</td>
</tr>
<tr>
<td>BorderStyle</td>
<td>DblClick</td>
<td>DoClick</td>
</tr>
<tr>
<td>Caption</td>
<td>Error</td>
<td>Refresh</td>
</tr>
<tr>
<td>Enabled</td>
<td>KeyDown</td>
<td>Stock Property Pages</td>
</tr>
<tr>
<td>Font</td>
<td>KeyPress</td>
<td>Colors</td>
</tr>
<tr>
<td>ForeColor</td>
<td>KeyUp</td>
<td>Fonts</td>
</tr>
<tr>
<td>hWind</td>
<td>MouseDown</td>
<td>Pictures</td>
</tr>
<tr>
<td>Text</td>
<td>MouseMove</td>
<td></td>
</tr>
<tr>
<td>*hybrid stock/custom method</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A property page is a dialog resource that allows the user to graphically view and edit a group of properties. The code for this dialog is completely contained within the control (not the container). Property pages are used mostly at design time, but they can also be displayed to users at run time. VBs do not have property pages: the onus is always on the container to manage the user interface for editing the control's properties, which is still allowed with OLE Custom Controls. The property page frame handles the display of the control's property pages, which includes an implementation of tabbed dialogs to switch between various pages. I'll talk more about property pages later.

As you saw at the bottom of Figure 5, the OLE services come from the OLE 2 DLLs. The OLE Custom Control architecture is based on standard OLE interfaces, so controls do not require the CDK's run-time DLLs. It is possible to construct controls that implement the control interfaces themselves and link directly to the OLE 2 DLLs. But this would be a lot of work and would defeat the purpose of the CDK, which is to insulate the programmers from having to write OLE code by providing MFC control classes that make it easier to construct OLE Custom Controls.

MFC Control Extensions

Central to the CDK are the MFC extensions that provide support for controls. The two important new base classes included in the CDK MFC extensions are COleControl, which implements the basic control interface support, and COlePropertyPage, which implements support for property pages. As illustrated in Figure 8, these base classes fit right in the class framework provided by MFC version 2.5 and greater.
Build Your Own OLE Custom Control

OK, now that you have a little background on OLE Custom Controls, let's build one using the CDK. I'll show you how to create a 16-bit control using Visual C++ version 1.5; the steps would change a little to create a 32-bit control using Microsoft Visual C++ version 2.0, primarily because the project management and build systems of Visual C++ 2.0 have been significantly improved over Visual C++ 1.5. For instance, Visual C++ 2.0 integrates MKTYPLIB, a standard OLE 2 utility that creates a type library, into the build system, which means that you will not have to run it as a separate step from the Tools menu. As mentioned earlier, an OLE Custom Control can have identical source code for both a 16- and 32-bit version, as long as it uses the MFC classes provided in the CDK.

I'll show all the steps required to make a simple "Smiley Face" OLE Custom Control, using the CDK. This control includes stock and custom properties, events, and methods and demonstrates most of the features of OLE Custom Controls and the CDK.

Installing the CDK

Since the CDK is an add-on to Visual C++, the CDK setup program installs the CDK as a subdirectory of your existing Visual C++ installation directory. If you accept the default, your CDK directory will be installed into the MSVC/CDK16 subdirectory (or CDK32 if you are installing the 32-bit components), except for the run-time DLL which is placed in the Windows system directory. The first difference you will notice is that the CDK setup program adds several entries to your Visual C++ Tools menu (see Figure 10). The App Studio entry was already added in this example by the Visual C++ 1.5 setup program; all the others are added by the CDK setup program; all the others are added by the CDK setup program.

Figure 10 CDK entries added to Visual C++ 1.5 Tools menu.

Figure 8 Abridged MFC Class Hierarchy for Controls

You benefit from this approach in a number of ways. First, your knowledge of MFC programming is leveraged onto controls. The programming model for controls is the standard MFC model, which means that you program using a familiar paradigm based on member functions and message maps. However, most controls, being small graphical components, simply don't require a document/view architecture.

With MFC control extensions, the full power of MFC is available for you to use in OLE Custom Controls. Also, your controls are now portable. The MFC extensions in the CDK give you the ability to maintain a single source-code base that can be compiled into a 16- or 32-bit control. As MFC and OLE 2 are ported to other new platforms, you can simply recompile your code for the new platform. Figure 9 describes all the new classes provided by the CDK.

Figure 9 New MFC Control Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Derived From</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CObject</td>
<td>CWnd</td>
<td>Base class for developing OLE custom controls. Derived from CWnd, this class inherits all the functionality of a window object plus additional functionality specific to OLE 2, such as event firing and the ability to support methods and properties.</td>
</tr>
<tr>
<td>CProperty</td>
<td>CDlg</td>
<td>Base class for developing property pages that display the properties of a custom control in the property page frame. Derived from CDlg, this class inherits all the functionality of a dialog object.</td>
</tr>
<tr>
<td>CControl</td>
<td>CWinApp</td>
<td>The base class from which you derive an OLE control module object. This provides members functions for initializing your control module (and each instance of it).</td>
</tr>
<tr>
<td>COleControlFactoryEx</td>
<td>COleObjectFactory</td>
<td>Extends the functionality of its base class COleObjectFactory by enforcing the OLE Custom Control licensing model and registering the control object factories with the OLE system registry.</td>
</tr>
<tr>
<td>CConnectionPoint</td>
<td>CWndTarget</td>
<td>Defines a special type of interface used to communicate with other OLE objects, called a &quot;connection point.&quot; A connection point implements an outgoing interface is able to initiate actions on other objects, such as firing events and change notifications. This is in contrast to a standard OLE interface, which passively exposes the functionality of an OLE control.</td>
</tr>
<tr>
<td>CPropExchange</td>
<td></td>
<td>The CPropExchange class and the classes derived from it (CResetPropExchange, CPropsetPropExchange, and CAchievePropExchange) support the implementation of persistence for your OLE controls. Persistence is the exchange of the control's state information, usually represented by its properties, between the control itself and a storage medium.</td>
</tr>
<tr>
<td>CFontHolder</td>
<td></td>
<td>Encapsulates the functionality of a Windows font object. Use this class to implement custom font properties for your control.</td>
</tr>
<tr>
<td>CPictureHolder</td>
<td></td>
<td>Implements a picture property. Use this class to create custom picture properties for your control which display bitmaps, icons, or metafiles.</td>
</tr>
</tbody>
</table>

Build Your Own OLE Custom Control

To build your own OLE Custom Control, you will need to:
1. Create a new control project.
2. Add the necessary MFC classes to your project.
3. Implement the necessary functionality in the control.
4. Register the control in the OLE system.

Figure 10 CDK entries added to Visual C++ 1.5 Tools menu.
As mentioned earlier, the ControlWizard creates initial control project files. MakeTypeLib is a standard OLE 2 SDK utility (MkTypLib.EXE) that compiles the Object Description Language (ODL) file into a binary type library (TLB). ODL is the source code version of a type library; a type library is the standard means by which an OLE 2 object describes itself.

Figure 11 is the ODL file for the finished Smiley control. It lists all the control's properties and methods in the primary dispatch interface, and all the control's events in the event dispatch interface. [Don't be confused by the methods: keyword that precedes the event list. Rather than making MktTypLib parse a new keyword for events, the methods: keyword was reused for the events dispatch map.] Figure 3 shows why the ODL file is arranged this way: the control has a single Dispatch interface for both properties and methods. This is what the container uses to communicate to the control. The container has an Dispatch interface for events, by which the control communicates to the container. A control's type library primarily describes these two interfaces. Also note that the type library describes more than just the external names for the properties, methods, and events. It also describes property types; return types; parameter names and parameter types for methods; and parameter names and parameter types for events. Finally, everything has a dispatch ID; stock items have standard predefined IDs; custom items have IDs generated by ClassWizard. The ControlWizard generates code that allows controls to automatically register themselves in the OLE 2 registration database.

Figure 11 Type Library Source (ODL) for the Finished Smiley Control

```cpp
// smile.odl : type library source for OLE Custom Control project.
// This file will be processed by the Make Type Library (mktplib) tool to
// produce the type library (smile.tlb) that will become a resource in
// smile.ocx.

#include <olectl.h>

[uuid(24C691E2-5D39-101B-83E0-0E939E9B295B), version(1.0),
 helpstring("Smile OLE Custom Control module")]
library Smile
{
  importlib(STDOLE_TLB);
  importlib(STDMETHOD_TLB);

  // Primary dispatch interface for CSmileCtrl
  [uuid(802BE1ED-73D0-101B-83E0-444553540000),
   helpstring("Dispatch interface for Smile Control")]
  dispinterface _DSmile
  {
    properties:
    // NOTE - ClassWizard will maintain property information here.
    // Use extreme caution when editing this section.
    [id(DISPID_FONT), bindable, requestedit]
    IFontDisp* Font;
    [id(DISPID_BACKCOLOR), bindable, requestedit]
    OLE_COLOR BackColor;
    [id(DISPID_FORECOLOR), bindable, requestedit]
    OLE_COLOR ForeColor;

    // NOTE - ClassWizard will maintain method information here.
    // Use extreme caution when editing this section.
    [AFX_ODL_METHOD(CSmileCtrl)]
    void Refresh();
    [AFX_ODL_METHOD]
    void AboutBox();

    // Event dispatch interface for CSmileCtrl
    [uuid(802BE1ED-73D0-101B-83E0-444553540000),
     helpstring("Event interface for Smile Control")]
    dispinterface _DSmileEvents
    {
      properties:
      // Event interface has no properties

      methods:
      // NOTE - ClassWizard will maintain event information here.
      // Use extreme caution when editing this section.
      [AFX_ODL_EVENT(CSmileCtrl)]
      void Click();
      [AFX_ODL_EVENT]
      void KeyPress(short* KeyAscii);
      [AFX_ODL_EVENT]
      void Outside();
    }

    // Class information for CSmileCtrl
    [uuid(24C691E2-5D39-101B-83E0-0E939E9B295B),
     helpstring("Smile Control")]
    class CSmile
    {
      [default] dispinterface _DSmile;
      [default, source] dispinterface _DSmileEvents;
    }
  }
}

The Register Control Tools menu entry invokes a small utility (REGSVR.EXE) that exercises this self-registration code. When the control is embedded in a container, the container automatically calls this self-registration code to register the control. The Unregister Control Command from the Tools menu calls the same registration utility, but passes it the /u switch to unregister the control; this removes it from the OLE 2 registration database.

Windows Process Status (WPS.EXE) is a standard SDK utility that allows viewing and modification of a process's vital statistics. WPS is useful during development for cleaning up after controls have managed to leave parts of themselves in memory after they've terminated. While your finished control will never do this, this could be a handy utility while you're developing. As mentioned earlier, the Test Container (TSTCON16.EXE or TSTCON32.EXE) allows you to test and debug your control's behavior.

The Control Wizard
It's no coincidence that **ControlWizard** (see Figure 12) looks very similar to the Visual C++ 1.5 **AppWizard**. Much of the technology that went into **AppWizard** was leveraged to create **ControlWizard**. The basic function of **ControlWizard**, like that of **AppWizard**, is to **generate** an initial MFC project; the difference is that **ControlWizard** generates all the necessary files to create an OLE Custom Control using the MFC Control Extensions provided in the CDK. **ControlWizard** allows you to specify the name and location of your control project and set various **options** for your control.

![Figure 12 Tools, ControlWizard](image)

The first three options in the Project Options dialog are **Context Sensitive Help**, **External Makelile**, and **Source Comments** (see Figure 13): they work exactly as their counterparts in **AppWizard**. The License Validation option is specific to controls and allows you, the control author, to determine who is able to use and distribute your control. If this option is checked, **ControlWizard** adds a small amount of source code to the control's H and CPP files and creates a license file (LIC). This essentially provides the same level of licensing that is available to a VBX, allowing you to supply the purchaser with the control and license file but legally allowing the purchaser to distribute only the control in his or her application. This prevents an application's users from using your control in other applications without first licensing it from you.

![Figure 13 ControlWizard Project Options](image)

The OLE Custom Control licensing model also allows alternative licensing methods that were not available to VBX authors. Since the CDK implements its licensing model with virtual member functions of your control's factory class, you can customize the licensing behavior of your control. For instance, you could override these functions in your control to provide different levels of licensing, exposing different properties, events, and methods based on the level of the license your control detected.

The Control Options dialog (see Figure 14) contains various features that you can enable or disable. **Activate** when visible, as the name implies, allows your control to be activated automatically when it is visible.

![Figure 14 ControlWizard Control Options](image)

Most OLE 2-aware container applications allow users to insert new OLE 2 objects by selecting the object from a standard Insert Object dialog. **The Show in Insert Object dialog option in ControlWizard** determines whether your control will be listed in container applications' Insert Object dialogs.

**The Invisible at run-time option** determines whether your control is displayed at run time. This allows certain specialized controls that have no run-time display, such as timer controls, to remain invisible at run time.

**The Simple Frame option** is used to create controls capable of operating as simple frames controls that can be used to group other controls in containers that support them. There are two primary benefits: first, you get the ability to visually group controls, for instance you can place radio buttons within a group box. Second, the simple frame allows you to nest HWNDs without requiring you to have a full OLE container.

When checked, the About box option creates a standard About box for your control, along with an About method.

The **Subclass Windows control** option allows you to create a custom control that is subclassed from a standard Windows system control, such as push buttons, scrollbars, and so on.

The **Use VBX control as template** option lets you use an existing VBX control as the template for your new OLE Custom Control. Once a VBX control is selected, the ControlWizard loads it, looks at its interfaces, and creates similar stub interfaces in the OLE Custom Control. It is a simple mechanism: the interfaces are copied, the implementation is not. This option will copy all VBX custom properties and custom events and most VBX standard properties and events. A few of these standard properties aren't supported by the control. For instance, VBX standard properties like Left, Top, Height, and Width are treated as extender properties in OLE Custom Controls. The container is responsible for supplying them, not the control. This option doesn't copy methods, but this shouldn't be a real limitation since VBXs had few standard methods and custom methods were not allowed.
The Controls dialog (see Figure 15) allows you to view and change any class names, filenames, user names, or IDs for either the class or the property page class. Any names on this dialog can be individually edited, or you can change them all at once by changing the name in the Short Name edit box. The Controls list box has plenty of room for more than one control. That's because this is also the dialog you can use to add controls to your OLE Custom Control. Just like VBXs, an OCX file can contain more than one control. As shown in Figure 15, this is supported in the ControlWizard when you initially generate your project, but it's also easy to add new controls manually after the project has been created. Clicking the Add Control button adds a new control to the project; the name for that control can also be set by modifying its Short Name string.

![ControlWizard Controls dialog.](image)

Figure 15 ControlWizard Controls dialog.

After the OK button is pressed on the main MFC ControlWizard dialog, a confirmation dialog is displayed (similar to the one in AppWizard) summarizing the features and names of the control about to be created. Once the Create button on that dialog is pressed, ControlWizard creates the files and loads the project into the Visual C++ IDE. ControlWizard automatically generates a README.TXT file, which describes all the files it creates (see Figure 16).

**Figure 16 README.TXT Automatically Generated by ControlWizard**

OLE Custom Control DLL: SMILE

=================================================================================================

ControlWizard has created this project for your SMILE OLE Custom Control DLL, which contains 1 Control. This skeleton project not only demonstrates the basics of writing an OLE Custom Control, but is also a starting point for writing the specific features of your control.

This file contains a summary of what you will find in each of the files that make up your SMILE OLE Custom Control DLL.

**SMILE.MAK**

The project makefile for building your 16-bit OLE Custom Control. This project file is compatible with the Visual C++ 1.5 Workbench. It is also compatible with NMAKE.

**SMILE32.MAK**

The project makefile for building your 32-bit OLE Custom Control. This project file is compatible with the Visual C++ 2.0 Workbench. It is also compatible with NMAKE.

**MAKEFILE**

A makefile that makes it easy to run NMAKE from the command prompt. Use the following parameters with NMAKE:

- `DEBUG=0` Builds retail version
- `DEBUG=1` Builds debug version (default)
- `WIN32=0` Builds 16-bit version (default)
- `WIN32=1` Builds 32-bit version
- `UNICODE=0` Builds ANSI/DBCS version (default for WIN32=0)
- `UNICODE=1` Builds Unicode version (default for WIN32=1)

**SMILE.H**

This is the main include file for the OLE Custom Control DLL. It includes other project-specific includes such as RESOURCE.H.

**SMILE.CPP**

This is the main source file that contains the OLE DLL initialization, termination, and other bookkeeping.

**SMILE.RC**

This is a listing of the Microsoft Windows resources that the project uses. This file can be directly edited with the Visual C++ resource editor.

**SMILE.RC2**

This file contains resources that are not edited by the resource editor. Initially this contains a VERSIONINFO resource that you can customize for your OLE Custom Control DLL, and a TYPELIB resource for your DLL's type library. You should place other manually maintained resources in this file.

**SMILE.DEF**

This file contains information about the OLE Custom Control DLL that must be provided to run with Microsoft Windows. It defines parameters such as the name and description of the DLL, and the size of the initial local heap. The numbers in this file are typical for OLE Custom Control DLLs.

**SMILE32.DEF**

This is a version of SMILE.DEF for when building a 32-bit version of the OLE Custom Control DLL.

**SMILE.CLM**

This file contains information used by ClassWizard to edit existing classes.
The code for the Smiley Control

```cpp
// This file contains the declaration of the CSmileCtrl class.

// This file contains the implementation of the CSmileCtrl class.

// This file contains the implementation of the CSmilePropPage class.

// This file contains the implementation of the CSmilePropPage class.

// Other standard files:

// This is the standard header file, which defines new resource IDs.

// This is included by the main resource file SMILE.RC.

// Other notes:

// ControlWizard uses "TODO:" to indicate parts of the source code you should add to or customize.

// The two files that matter most are SMILECTL.CPP, where the bulk of the implementation for the Smiley control resides in the member functions of the CSmileCtrl class; and SMILEPPG.CPP, where the Smiley property page is implemented in the member functions of the CSmilePropPage class.

At this point I have used ControlWizard to automatically generate 22 files that will compile cleanly the control just won't do anything interesting yet.

OnDraw

If you're like most developers, you want to see results quickly. I'll put off adding properties, events, or methods for the moment while I add some drawing code that lets me see something resembling a smiley face.

First I'll open the SMILECTL.CPP file and add the lines of code shown in Figure 17 to the OnDraw member function of the CSmileCtrl class. Then I'll set up some scaling macros to make sure my drawing works in coordinates from 0.0 at the upper-left corner to 100.100 at the lower-right corner. This allows the drawing code to always work in the same coordinate space regardless of the actual pixel size of the control. If the control is resized in either direction, the drawing code will always scale correctly. The pen width is also based on the scaling macros so that the thickness of the lines drawn in the smiley face are proportional to the size of the control. Following that, there are CRect calls using brushes and pens to draw the ellipses and arcs that make up the smiley face.

Figure 17 Initial Display Code for the Smiley Control

```cpp
void CSmileCtrl::OnDraw(
    CDC* pdc, 
    CRect& rcBounds, 
    const CRect& rcInvalid) 
{ 
    long xLeft = rcBounds.left; 
    long yTop = rcBounds.top; 
    long xScale = rcBounds.Width(); 
    long yScale = rcBounds.Height(); 
    int iPenWidth = max(CX(5), CY(5)); 
    CPen *penSave = pdc->SelectObject(&penNull); 
    CBrush *pBrushSave = pdc->SelectObject(&brushBlack); 
    pdc->Ellipse(X(25), Y(10), X(75), Y(80)); // Smile mouth
    pdc->Ellipse(X(35), Y(15), X(65), Y(90)); // Right eye
    pdc->Ellipse(X(35), Y(35), X(43), Y(50)); // Left eye
    pdc->SetPixel(X(30), Y(30), 0x000000); // Null pen for drawing filled ellipses
    pdc->FillRect(rcBounds, &brushBack); // Background
    pdc->SelectObject(&penNull); 
    CRect rcClip = rcInvalid; 
    void CSmileCtrl::OnDraw( 
        CDC* pdc, 
        CRect& rcBounds, 
        const CRect& rcInvalid) 
    { 
        int iPenWidth = max(CX(5), CY(5)); 
        // Pen width based on control size
        CBrush* pBrushBlack = pdc->SelectObject(&brushBlack); 
        CBrush* pBrushYellow = pdc->SelectObject(&brushYellow); 
        // Background
        // null pen for drawing filled ellipses
        pdc->FillRect(rcBounds, &brushBack); // Clear background
        CPen* pPenSave = pdc->SelectObject(&penNull); 
        CBrush* pBrushSave = pdc->SelectObject(&brushBlack); 
        pdc->Ellipse(X(25), Y(10), X(75), Y(80)); // Smile mouth
        pdc->Ellipse(X(35), Y(15), X(65), Y(90)); // Right eye
        pdc->Ellipse(X(35), Y(35), X(43), Y(50)); // Left eye
    }
    void CSmileCtrl::OnDraw( 
        CDC* pdc, 
        CRect& rcBounds, 
        const CRect& rcInvalid) 
    { 
        int iPenWidth = max(CX(5), CY(5)); 
        // Pen width based on control size
        CBrush* pBrushBlack = pdc->SelectObject(&brushBlack); 
        CBrush* pBrushYellow = pdc->SelectObject(&brushYellow); 
        // Background
        // null pen for drawing filled ellipses
        pdc->FillRect(rcBounds, &brushBack); // Clear background
        CPen* pPenSave = pdc->SelectObject(&penNull); 
        CBrush* pBrushSave = pdc->SelectObject(&brushBlack); 
        pdc->Ellipse(X(25), Y(10), X(75), Y(80)); // Smile mouth
        pdc->Ellipse(X(35), Y(15), X(65), Y(90)); // Right eye
        pdc->Ellipse(X(35), Y(35), X(43), Y(50)); // Left eye
        pdc->Ellipse(X(57), Y(35), X(65), Y(50)); // Left eye
        pdc->SelectObject(pBrushSave); 
        pdc->SelectObject(pPenSave); 
    }
```
the constructor in the control class. For the Smiley control, I'll set up an initial square size of 200 by 200 pixels:

```cpp
CSmileCtrl::CSmileCtrl()
{
    SetInitialSize(200, 200);
    InitializeIIDs(
        &IID_DSmile,
        &IID_DSmileEvents);
}
```

Building the control takes just three steps when using Visual C++ version 1.5: run Tools.Make TypeLib to compile the ODL file, select Project Build to compile the control, and run Tools.Register Control to add it to the OLE 2 registry. Then run the Test Container and insert the control.

Figure 18 shows my current Smiley control inserted into the Test Container. At this point it's not muchthe graphical equivalent of "hello, world." I still need to add properties, events, and methods to really show off the power of controls, but now I have the core. I can insert it into a container, display, activate, resize, move, and even show an About box for it. In fact, I've already written about half the code needed to complete the control.

Adding Properties

I'll be adding five properties to the Smiley control: BackColor, Caption, Font, ForeColor, and Sad. The first four are stock properties; Sad is a custom property. The COleControl class contains predefined member functions that are used to support stock properties, usually one to get the property and one to set the property. The control author simply selects the stock property in the ClassWizard Add Property dialog and all the "plumbing" work is done. All you need to do is use the stock property's member functions to access its values.

The first stock property I'll add is Caption. This will display a user-defined caption above the smiley face. To begin I select Browse ClassWizard from the IDE menu and then select the OLE Automation page (see Figure 19), which provides the support for properties and methods. You'll need to make sure that the control class (in this case CSmileCtrl) is selected, not the property page CSmilePropPage class. Next you click the Add Property button and select Caption from the list of stock properties in the External Name drop-down combo box (see Figure 20).

---

Figure 18 Simple Smiley face control in the Test Container.

Figure 19 ClassWizard OLE Automation page.
Click OK and you've just added the Caption stock property to the Smiley control. The ClassWizard creates the property for you, but you still have to code up the implementation of what the control does based on that property. In this case it's a matter of writing a few new lines of code in the OnDraw function to display the caption string above the smiley face (see Figure 22, which I'll discuss later). For now you need to repeat the procedure I just used for the Caption property to add the ForeColor, BackColor, and Font properties. I'll also add some code to wire up these properties in the OnDraw function later. I need to do just a bit more work for the custom property Sad, but not much.

Implementing Properties

In the Add Property dialog of Figure 20 you can see three radio buttons grouped together under Implementation. The Stock implementation is used when you select a stock property; this automatically assigns the correct MFC Get and Set member functions. For instance, the Caption stock property is accessed via the GetText and SetText functions and is stored behind the scenes as a char pointer to a character (BSTR). The Member Variable implementation creates a member variable of the specified type in your control class to represent the property. It also automatically sets up a notification function for you. At run time the notification function is always called after the property value has changed. Your control can then do some action immediately after the property value changes, such as update its display or validate the value that was entered. If you don't need the notification function, you can delete it from the edt box and it won't be generated. The Member Variable-only implementation with no notification function has the least overhead. The Get/Set Methods implementation gives you the most control over access to the property. This implementation consists of a pair of member functions: the Get member function is automatically called when a caller requests the current value of a property; the Set member function is automatically called when a caller requests a change to the value of a property. This implementation can be used when you need to compute the value of a property at run time or validate an entered value before actually changing the property. This implementation can also be used to create read-only or write-only properties.

Adding Custom Properties

To add a custom property to the control, I need to once again click the Add Property button in the OLE Automation page of ClassWizard. When the Add Property dialog comes up, I type the property name, Sad, into the External Name field. ClassWizard automatically assigns the Variable Name field to m_sad, and the Notification Function field to OnSadChanged. The property type is set to BSTR since all I need to know is whether the property is on or isn't Sad. I click the OK button to add the custom property. Of course, I still have to write some code to perform actions based on the value of this property.

At this point the Name Field of ClassWizard's OLE Automation page displays all five newly added properties (see Figure 21). These properties are now in the CSmileCtrl class, but they won't do anything until I add code that performs some action based on the values of the properties. I'll add code to the OnDraw function that changes the control's display based on the property values (see Figure 22).

Figure 20 Adding a stock property to the Smiley control.

Figure 21 Smiley control properties.

Figure 22 Changing the Display Based on Properties

```c++
void CSmileCtrl::OnDraw(CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    int iPenWidth = max(CX(5), CY(5));        // Pen
    CBrush brushBack;                   // Stock
    CBrush brushYellow;                         // Background
    CPen pPenSave = pdc->SelectObject(&penBlack);
    CBrush* pBrushSave = pdc->SelectObject(&brushYellow);
    pdc->FillRect(rcBounds, &brushBack);                   //Stock
    // Add brushFillRect
    // Add brushGetColor
    // Add draw functions
    // Add button functions
    // Add text functions
    // Add graphic functions
    return;
}
```

### CRect& rcBounds, const CRect& rcInvalid)

```c++
void CSmileCtrl::OnDraw(C
```
pdc->Ellipse(X(10), Y(15), X(90), Y(95));  // Head
if (m_sad)
   return; //CustomProperty
pdc->Arc(X(25), Y(70), X(75), Y(140),
   X(45), Y(70), X(35), Y(75));
} else {
   pdc->Arc(X(25), Y(1D), X(75), Y(80),
   X(35), Y(70), X(65), Y(70));
}
pdc->SelectObject(penNull);  // No draw width
pdc->SelectObject(brushBlack);
pdc->Ellipse(X(35), Y(35), X(43), Y(50));  // Right eye
pdc->Ellipse(X(57), Y(35), X(65), Y(50));  // Left eye
pdc->SetBkMode(TRANSPARENT);  // Use ForeColor
pdc->SetTextColor(pdc);  // Use Font
pdc->SetStockFont(pdc);  // Stock Property
pdc->DrawText((InternalGetText()), -1, rcBounds,      // Use Caption
   DT_SINGLELINE | DT_CENTER | DT_TOP);  // Stock Property
pdc->SelectObject(pBrushSave);
pdc->SelectObject(pEndSave);
}

To wire up the BackColor stock property, I'll first use the GetBackColor function to retrieve the background color value. Then I'll create a brush for painting the background based on that color. Finally I'll comment out the old line of code that always creates a white background brush, when I do that a few lines later, I'll be using the BackColor property value to paint the control's background. To implement the Sad custom property, I'll set up an if statement that uses an Arc function to either draw a frown or a smile depending on the value of m_sad. (Recall that m_sad is the member variable for the Sad custom property.) For the ForeColor stock property, I pass the return value of the GetForeColor function to SetTextColor so that any text drawn will use the color specified in the ForeColor property. I use the SelectStockFont function to set the font for the device context so it matches the font specified by the Font property. The ForeColor and Font properties are used to generate modifiers for the Caption property. The Caption property string is accessed by a call to InternalGetText, I pass this to DrawText, which displays the Caption string at the top of the control using the current ForeColor and Font. InternalGetText is a documented variant of GetText that gets called automatically whenever the Caption property changes. Now I'll compile and run the control, I can modify its properties to make it look something like Figure 23.

Figure 23 Smiley with Sad= True, BackColor= Blue, ForeColor= Magenta, Font= 36pt Italic Bold Book Antiqua, Caption= “Sad Face.”

Property Pages

Now all the properties are implemented and the OnDraw code is set up to display based on the property values. It's a good time to talk about how a user can view and modify the actual values of a control's properties. One method (see Figure 24) has the container application query the control via the Properties (Dispatch) interface to get the property values and then provide a user interface to access the current property values. VBX controls only allow user access to properties via the container. The second method (see Figure 25) requires the control author to implement one or more property pages for the control. The container can request the control to display its property pages by issuing the Properties verb (OLEIVERB_PROPERTIES) to the control. The Test Container supports both methods, I used the first to set the properties for the image in Figure 23 since I hadn't added any property pages, but this method relies on the container to provide the user interface. With this method you have no control over how the properties are displayed. In the case of the Test Container, the interface is fairly simplistic. My recommendation is for the control author to decide how the user will interact with the control's properties and provide property pages that enable this interaction.

Figure 24 Container-managed property access.
A property page starts out as a dialog resource, and ControlWizard generates an initial blank property page. You can create more property pages if you require them. Because they’re dialog resources, it’s easy to use App Studio to lay out the user interface for the control properties on the property page (see Figure 26). You can also use any of the stock property pages, which are implemented for you in the control run-time DLL. There are stock property pages for Colors (see Figure 25), Fonts, and Pictures. Any properties you create that are of one of these three types can be linked to the corresponding stock property page by adding one line of code to the control, which I’ll show later.

First I’ll show how to use App Studio to manually lay out the initial property page dialog that ControlWizard automatically gives us. To do that you need to start App Studio from the Visual C++ Tools menu. Selecting the Dialog resource type and the IDD_PROPPAGE_SMILE dialog will bring up a blank dialog corresponding to the control’s General property page. Control Wizard assigns a default name of General to custom property pages. You can easily change the property page name by editing the corresponding string resource in App Studio.

Use the Control Palette to place controls on the dialog. Place a static text control named Caption: and next to it an edit text control. Double-clicking on the edit control brings up the property editor, which you can use to enter IDC_CAPTION as the control’s ID. Below the Caption text place a check box called Sad, with ID: IDC_SAD. At this point your window should look like Figure 26. Nothing forced me to use a check box to represent the Sad property; I could just as easily have used two radio buttons (Smile and Frown) or even an edit box where the user had to type in Happy or Sad. You can create the property page user interface however you wish. Of course you should try to stay within the bounds of standard UI design guidelines as specified by the Windows User Interface Design Guide.

That takes care of the layout for the General property page, which includes the Caption and Sad properties. I’ll use stock property pages for the ForeColor, BackColor, and Font properties.

First I need to use ClassWizard for two more important tasks: adding property page member variables and adding the property transfer code. I already have property member variables set up in the CSmileCtrl class to hold the values in the control, but I also need temporary variables to hold the values that are entered on the property page. These member variables are in the CSmilePropPage class and provide temporary storage for the property values entered in the property page. Once the values are committed by clicking the OK or Apply buttons on the property page, the values are transferred to the control’s properties.

The property transfer code is provided by DDX/DDP functions that synchronize property values between the control and property page. Dialog data exchange (DDX) is the standard MFC method for doing data transfer between the control, in a dialog box or form view and the data members in a corresponding class. DDX gets the property from the Windows control on the property page’s dialog into the property page’s (CSmilePropPageClass) member variable. Property page data transfer (DDP) is new in the CDK and is used to transfer the value from the property page member variable to its implementation in the OLE Custom Control (CSmileCtrl) by linking the property page’s data member to the property’s external name. To add in the DDX/DDP functions, I need to launch ClassWizard. Since I just edited the General property page in App Studio it makes more sense to start ClassWizard from there. I’ll select the Member Variables page in ClassWizard and pick CSmilePropPage from the Class Name list box. First I’ll add the member variable for the Caption property’s text edit box by selecting the IDC_CAPTION Control (see Figure 27).

Figure 25 The stock color property page

Figure 26 Editing the dialog resource for the General property page.
Figure 27 The ClassWizard Member Variables page.

Clicking on the Add Variable button brings up the Add Member Variable dialog (see Figure 28). I type `m_caption` as the Member Variable Name and make sure that the Variable Type is `CString`. I select Caption from the Optional OLE Property Name drop-down combo box. This lets ClassWizard know I want to hook this property page member variable up to the Caption property in the control, and it's what causes the DDP code to be generated in the CSmilePropPage:DoDataExchange function. I click OK, and I'm done with the Caption property. Adding a member variable for the Sad property is a similar process. I select the IDC_SAD Control ID in ClassWizard and click the Add Variable button. I type `m_sad` as the Member Variable Name and make sure that the Variable Type is `BOOL`. Since Sad is a custom property, I need to type Sad into the Optional OLE Property Name combo box to link it up with the control's Sad property. I click OK, then I'm done with both variables.

Figure 28 Adding a Member Variable in the property page.

Now we have our property page member variables defined and ClassWizard has automatically written DDX/DDP property transfer code in the DoDataExchange function of SMILEPPG.CPP:

```cpp
// CSmilePropPage::DoDataExchange - Moves data between page and properties
void CSmilePropPage::DoDataExchange(CDataExchange* pDX)
{
//{{AFX_DATA_MAP
    AFX_DATA_MAP(CSmilePropPage)
    DDP_Text(pDX, IDC_CAPTION, m_caption, _T("Caption"));
    DDX_Text(pDX, IDC_CAPTION, m_caption);
    DDP_Check(pDX, IDC_SAD, m_sad, _T("Sad"));
    DDX_Check(pDX, IDC_SAD, m_sad);
//}}AFX_DATA_MAP
    DDP_PostProcessing(pDX);
}
```

As mentioned earlier I need to add a line of code for each stock property page I enable. Since I'm using the color and font stock property pages, I'll add the following two lines of code to the property page table in SMILECTL.CPP (and of course change the property page count from 1 to 3):

```cpp
BEGIN_PROPPAGEIDS(CSmileCtrl, 3)
PROPPAGEID(CSmilePropPage::guid) //Stock Color Property
PROPPAGEID(CLSID_CColorPropPage) //Stock Color Property
PROPPAGEID(CLSID_CFontPropPage) //Stock Font Property
END_PROPPAGEIDS(CSmileCtrl)
```

Together, these macros contain code that initializes an array of Class IDs that represent the property pages. After compiling the control and loading it up in the Test Container, I can look at the new General property page and use it to change the values for the Caption and Sad properties (see Figure 29).

I can also use the stock property pages I enabled to change the ForeColor, BackColor, and Font properties. Figure 25 shows the Color stock property page, and Figure 30 shows the Font stock property page. Note that the stock property pages come equipped with a Property Name list box that allows selection of the specific property to edit. In the Color property page, I can use this list box to select either the ForeColor or BackColor property.
Initialize and Be Persistent

You may wish to have your properties initialized to some specific value when an instance of your control is first created. You may also want your properties to be able to retain their states even after they are no longer running in a container. This is referred to as persistence (also, as serialization). Persistence allows your controls to have “memory” so that when your application is run the next time, any controls in it have the same values they last had when the application was shut down. All the stock properties have initialization and persistence support already enabled, but any custom properties you might add will not have this support. If I compile and run the Smiley control at this point, it will work fine each time I run it, but it will come initially with the Sad property set, yielding a sad face. Also, the Smiley control won’t retain the value of its Sad property between runs. I can take care of this problem with one additional line of code added to the DoPropExchange function in SMILECTL.CPP (the DoPropExchange function is where all of the control’s initialization and persistence code is placed):

```cpp
// CSmileCtrl::DoPropExchange - Persistence support
void CSmileCtrl::DoPropExchange(CPropExchange* pPX)
{
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
    COleControl::DoPropExchange(pPX);
    // TODO: Call PX_ functions for each persistent custom property.
    PX_Bool(pPX, "Sad", m_sad, FALSE); //Init & Persistence
}
```

The Big Event(s)

Just like properties, events come in both stock and custom flavors (see Figure 7 for the list of stock events). I’ll add four events to the Smiley control: Click, KeyPress, Inside, and Outside. The first two are stock events, the last two are custom events. The Click stock event is generated when the user clicks a mouse button, and the KeyPress stock event is generated when the user types a key. I’ll make the control generate an Inside custom event when the user clicks the left mouse button while the mouse pointer is inside the Smiley control’s ellipse. To demonstrate parameterized events, I’ll also pass the coordinates for the x and y location of the mouse pointer at the time the Inside event occurs. If the mouse pointer is outside of the Smiley control’s ellipse when the user clicks the left mouse button, I’ll make the control generate...
an Outside event instead. To add events, I need to once again bring up the ClassWizard, this time selecting the OLE Events page and making sure that CSmileCtrl is selected in the Class Name list box (see Figure 32).

Figure 32 The ClassWizard OLE Events page.

The first event I'll add is Click. Clicking the Add Event button brings up the Add Event dialog (see Figure 33). Since this is a stock event, I just need to select Click from the External Name drop-down combo box and hit the OK button to complete the process. To add the KeyPress stock event I'll go through the same steps, this time selecting KeyPress from the External Name drop-down combo box.

Figure 33 Adding a stock event to the Smiley control.

Next I'll add the custom events Inside and Outside. My strategy: when the user clicks the left mouse button, check to see if the mouse pointer is inside the ellipse of the Smiley control. If it is, I want to call the Inside event firing function (FireInside) and pass the x and y coordinates as parameters. If it's not inside I want to call the Outside event firing function (FireOutside). To do this, I'll use a standard MFC Message Map to map the WM_LBUTTONDOWN message. Once again I click the Add Event button in ClassWizard's OLE Events page to bring up the Add Event dialog (see Figure 34); I type Inside into the External Name field and see that ClassWizard automatically assigns the name FireInside to the Internal Name (the firing function).

Figure 34 Adding a custom event with parameters.

The Inside event will be a parameterized event, with parameters for the x and y mouse coordinates at the time of the mouse click. To add parameters, I first click the Add button to bring up the Add Event Parameter dialog (see Figure 35). I type X into the Name field for the x coordinate and select short as the Type. Then I click the Add button to add the parameter. I do it again for the y parameter and click the Close button to return to the Add Event dialog. I repeat the process to add the Outside event, typing Outside as the event’s External Name, but I don’t add any parameters for the Outside event. ClassWizard’s OLE Events page now lists all the stock and custom events I’ve added (see Figure 36).

The stock events are completely implemented, but for the custom events I’ll hook up an actual user action to the firing functions (FireInside, FireOutside). I select the Message Maps page in the ClassWizard (see Figure 37). CSmileCtrl should be selected as the Class Name. Then I
click on the CSmileCtrl Object ID, select the WM_LBUTTONDOWN message, and click the Add Function button to add the OnLButtonDown function to the CSmileCtrl class in the SMILECTL.CPP file. I click the Edit Code button to fill in the function.

![Figure 35 AddEvent Parameter Dialog.](image)

![Figure 36 Stock and custom events list.](image)

![Figure 37 ClassWizard's message maps page.](image)

OnLButtonDown is called whenever a WM_LBUTTONDOWN is generated, which occurs when the user clicks the left mouse button. When that happens, I check to see if the mouse pointer is inside of the Smiley control's ellipse. If it is, I fire the Inside event via the FireInside function, passing it the x and y coordinates of the mouse pointer. Otherwise I fire the Outside event via the FireOutside function with no parameters. The xLeft, yTop, xScale, and yScale variables are required by the X and Y scaling macros I used in the OnDraw function. I'll use the scaling macros again now to adjust the parameters I'm passing to InEllipse:

```cpp
void CSmileCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rcBounds;
    GetClientRect(&rcBounds);
    long xLeft = rcBounds.left; //Required by X() and Y() macros
    long yTop = rcBounds.top;
    long xScale = rcBounds.Width();
    long yScale = rcBounds.Height();
    if (InEllipse(point, X(10), Y(15), X(90), Y(95)))
        FireInside(point.x, point.y);
    else
        FireOutside();
    COleControl::OnLButtonDown(nFlags, point);
}
```

InEllipse is a function I wrote in the SMILECTL.CPP file to do elliptical hit-testing. When passed a point and the boundary of an ellipse, InEllipse returns TRUE if the point is within the ellipse, FALSE if it isn't:

```cpp
BOOL CSmileCtrl::InEllipse(point, int x1, int y1, int x2, int y2)
{
    // Determine radii
    double a = (x2 - x1) / 2;
    double b = (y2 - y1) / 2;
    // Determine x, y
    double x = pt.x - (x1 + x2) / 2;
    double y = pt.y - (y1 + y2) / 2;
    // Apply ellipse formula
    return ((x * x) / (a * a) + (y * y) / (b * b) <= 1);
}
```

I also need to add InEllipse to the protected section of the CSmileCtrl class definition in the SMILECTL.H file:

```cpp
class CSmileCtrl : public COleControl
{
```

Building Component Software With Visual C++ and the OLE Custom Control Developer's Kit  
(C) 1992-1995 Microsoft Corporation. All rights reserved.
After compiling this version of the control, I can start up the Test Container and select View Event Log to test the new events (see Figure 38). Whenever a left mouse click is made anywhere on the control, a Click event is generated. When a left mouse click is made inside of the Smiley ellipse, the Inside(x, y) custom event, is generated. When a click is made outside of the Smiley ellipse, the Outside stock event with no parameters is generated. Finally, when a key is pressed, the stock KeyPress(nnn) event with the ASCII key code as a parameter is generated. Since the Test Container is able to support more than one control at a time and multiple versions of the same control, it needs to be able to properly identify which events come from which control. That's why the index and control name are added to the front of the event in the Event Log, for example 00_Smile_Control in Figure 38.

Figure 38 Testing events.

Add a Method to the Madness

I'll add two methods to the Smiley control: Refresh and Beep. Refresh is a stock method that causes the control to repaint itself; Beep is a custom method that calls the Windows MessageBeep function. Since MessageBeep takes a parameter, this also gives me an opportunity to demonstrate how to add a parameter to a method. You've probably guessed by now that the first step is to start ClassWizard and select the OLE Automation page (see Figure 19). To add the Refresh stock method, click on the Add Method button, which brings up the Add Method dialog (see Figure 39). I select Refresh from the External Name list box, click OK, and it's done.

Figure 39 Add Method dialog.

To add the Beep custom method, click the ClassWizard Add Method button again. This time I type Beep in the External Name field of the Add Method dialog and select void as the Return Type (since MessageBeep returns a void). I click the Add button to bring up the Add Method Parameter dialog (see Figure 40). I type uAlert as the Name of the parameter and select short as the Type. The uAlert parameter in MessageBeep is actually a UINT, but I'll just cast the short to the UINT when I pass it. I click the Add button to add the parameter, and the Close button to return to the Add Method dialog. I click the OK button on the Add Method dialog and you'll be back in ClassWizard. I've added both new methods for the control now; I just need to wire up the custom method so I can do something interesting. In this case I'll keep it simple and just have Beep call the Windows MessageBeep function, passing it the uAlert parameter. Click the Edit Code button on ClassWizard, which takes you to the Beep function in the SMILECTL.CPP file. Then I add the following:

```cpp
void CSmileCtrl::Beep(short uAlert)
{ MessageBeep((UINT)uAlert); }
```

Now I can compile the control and test the methods in the Test Container. When I launch the Test Container, insert the Smiley control, and select Invoke Methods from the Edit menu, I get the Invoke Control Method dialog, which in this case displays three dropdown list boxes: one for the Name of the method, one for the method ID, and one for the uAlert parameter. To test the Beep method, I select Beep from the Name list box, type in a value for the uAlert parameter (such as 0), and click the Invoke button and voilà, my machine beeps. The Refresh and About box methods can be tested in the same way.

The ControlWizard creates a default bitmap that containers can install in their toolbars. You can edit the bitmap to make it appropriate for your own control. I used App Studio to edit the IDB_SMILE bitmap (see Figure 41).
After editing the bitmap, you can rebuild the control and use the Test Container to try it out. The Test Container lets you test out the bitmap by installing it into its very own toolbar (see Figure 42). Finally, you may wish to use App Studio to customize the default About box that ControlWizard generates.

Figure 42 Testing the toolbar bitmap in the Test Container.

Conclusion

There you have it: about 70 lines of user-written code (a 32K binary) to create a full-fledged albeit sillyOLE Custom Control. I didn’t show you the hundreds of lines of code generated automatically by ControlWizard to create the initial Smile project, or the thousands of lines of code in MFC that made this possible. The Smiley control exercises most of the major features of the CDK except for data binding, licensing, versioning, internationalization, the VBX coding utility, and Windows control subclassing.

Because OLE Custom Controls are based on MFC and OLE 2, they can be compiled to work on both 16- and 32-bit Windows platforms and are ready for other platforms that will support OLE 2 and MFC in the future.

OLE Custom Control technology is also at the heart of future Microsoft operating systems such as Cairo, which means that over time the control run-time DLL functionality will be built right into the operating system. As was the case with VBXs, it is natural to expect that future versions of the OLE Custom Control architecture will evolve to support new features. Finally, it’s probably a safe bet that OLE Custom Control support will be more thoroughly integrated into future versions of Visual C++ and become even easier to use.