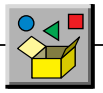

OLE EXPERT

Tackling OLE Control Problems

Find out everything you ever wanted to know about OLE control installation, but were afraid to ask.

by Sam Patterson

Let me start by saying that this column is not meant to be a commercial endorsement for the company I am currently working for. There are many great third-party component companies that provide a wide breadth of add-on and add-in products. Many of these companies have overcome the same obstacles that we faced when creating, testing, and distributing our OLE controls for Visual Basic 4.0. Much of the information in this article is basic information that I have gathered over the past three years working with OLE controls. So even though it may seem basic at first, you should find something in this column that will be useful.

My first experience programming OLE controls was for Microsoft. I wrote nine controls for the Visual Basic group and seven are included in the Visual Basic 4.0 box. It was quite an experience working directly with the Visual Basic developers on this project, and I learned many new tricks and techniques for improving the performance and understanding the whole architecture of OLE controls. Because I wasn't directly involved in most of the testing, installation, setup, and support issues involved with these controls, there were many things that I did not have a chance to learn from the experience. I was to learn very quickly, however! <g> Since joining MicroHelp as general manager of the Component Product Division in July 1995, I have learned more about OLE controls than I would have just by programming them alone or from the Microsoft experience. In the process of creating many different retail packages of OLE controls for MicroHelp, many issues have come up when programming, installing, using, and supporting OLE controls that are appropriate for the OLE Expert column. In this month's column, I will explain all of the hurdles we overcame in creating various retail packages with a special emphasis on the installation of the controls under various platforms. I also will discuss the support issues that we have faced once the controls made it onto users' systems. Many of these issues will become more and more important to Visual Basic programmers as the next version of Visual Basic has been purported by the trade press to create OLE controls.

Sam Patterson is general manager of the Component Products Business Unit of MicroHelp Inc. and a contributing editor of Visual Basic Programmer's Journal. *He is also owner of Gold Leaf Systems, a Los Angeles, California-based consultancy specializing in VBX/OLE Control component software development. He is coauthor of MicroHelp's* OLETools, VBTools, SpellPro, Thesaurus, *and V*BComm 3.0 Communications Library. *He is also the author of the MCI, MAPI, Masked Edit, Rich Text, Toolbar, and TabStrip OLE controls included with Microsoft's Visual Basic 4.0, Visual FoxPro 3.0, and Visual C++ 4.0. Contact Sam at MicroHelp Inc., by e-mail at SamP@microhelp.com or by fax at 404-645-2122; or at Gold Leaf Systems, by mail at 5301 Beethoven Street #190, Los Angeles, CA 90066-7061 or by fax at 310-574-6301. Reach Sam on CompuServe at 72000,1751.*

ARCHITECTURE AND FILES

First, let's talk a little about the OLE control architecture. OLE controls are like any other OLE objects. They are programmable objects that implement certain functionality that they expose through various OLE interfaces. One of the first obstacles to overcome when installing and using an OLE control is to make sure that all of the supporting files, such as the OLE DLLs, language DLLs, and so forth that are needed to allow the actual control to execute are present. For example, if you build your 32-bit controls using Visual C++ 4.0 using MFC and you selected the option to use MFC40.DLL, you have to distribute MSVCRT40.DLL and MFC40.DLL on your installation disks. MSVCRT40.DLL is the runtime DLL for the C++ programs and MFC40.DLL is the foundation class runtime library. You also need to distribute any other DLLs that your control uses.

For example, in OLETools we have a runtime DLL that encapsulates many of the common routines used across our controls called MHRUN32.DLL. This has to be distributed and installed before any of our controls can operate. The most up-to-date OLE DLLs should also be distributed. This includes (but is not limited to) OLE32.DLL, OLEAUT32.DLL, OLECLI32.DLL, and so forth. We have found that having our OLE controls running on older builds of the OLE DLLs can cause many problems ranging from GPFs to very odd runtime behavior. If you are distributing a 16-bit OLE control, you need to distribute the correct 16-bit MFC DLLs, along with OC25.DLL, the DLL that implements the 16-bit version of OLE custom controls. You must also make sure that the OLE DLLs, MFC DLLs, and the OC25.DLL are registered. If they are not registered your control will neither register nor load.

The next step before you can actually use the control is to "register" the control. OLE controls and other OLE objects use the registry to record information about the location of their executable, help file, and so forth. These are all tied together using a unique number called a Class ID (CLSID). OLE uses this number, or key, to create the objects. Most OLE-enabled applications are "self registering." This means that the first time you run the application they register all the information about the OLE components that the application provides. OLE controls are somewhat different. Because OLE controls are not EXEs—they are actually DLLs—there is no way for them to "automatically" self-register when they run. To overcome this problem, most OLE controls have two functions internally that can be called to register all the components that the control provides. Any language that supports DLLs can call these functions, DLLRegisterServer and DLLUnregisterServer. Typical pseudo C code to load an OCX and call its register function looks something like this:

```
hmod = LoadLibrary("mhcmbo32.ocx");
```

```
lpfnRegister = GetProcAddress(hmod, _
   "DLLRegisterServer");
lpfnRegister();
```

When calling this function, the OLE control registers itself by writing its class information into the HKEY_CLASSES_ROOT section of the registry. It also adds a type library key into the appropriate section (the typelib section under HKEY_CLASSES_ROOT). A type library is a "standard" way of providing access to all of the names, parameter information, events, method names, and so forth for a specific OLE object. It also can contain constant names and values that programming languages such as Visual Basic can use. You can retrieve and use type library information from C as well as other applications that support the OLE type library interfaces. VB provides access to this through the object browser. The object browser shows what type libraries are referenced in your application. You also can use the object browser to look at all of the objects, methods, and other items contained within each referenced type library.

WHAT HAPPENED?

Once registered, the controls can be used in an OLE control container. However, even though registering the controls sounds easy and straightforward, many things can go wrong during the registration process. First and foremost relates back to the section where I explained that you need to distribute all DLLs and other files needed by the controls. This is especially true during the registration process. When you try to load an OLE control (as a DLL) using LoadLibrary, all files that are needed to load that DLL into memory must be present. For OLETools for example, if the MHRUN32.DLL is not present on the disk, then the LoadLibrary would fail. You therefore would not be able to register the control. The error messages returned from LoadLibrary do not provide a clue as to why the load failed either, so it can be difficult to determine what is going on. If any of the OLE DLLs, MFC DLLs, and so forth are missing, you get a cryptic error message that doesn't help you find the problem. So if you are trying to register the control, and you get the message back that it "can't load the library," or "LoadLibrary failed," make sure all your required files are there. Similar errors are generated if you are using the Microsoft REGSVR16 or REGSVR32 utilities. These generate a message box that is caused by trying to load an MhTree control without the MHRUN32.DLL present. As you can see, with this message box you might never figure out why you are unable to register the control (see Figure 1).

Other problems might arise when writing to the registry. For example, one such problem is corrupted registries. Although we have not seen this very often under 32-bit operating systems such as Win95 or NT, it has happened many times under Win 3.1 and Win 3.11. If something has corrupted the registry before you try to register your controls, some of your controls may register, and some may not. We have seen situations where half of the 16-bit controls register, and the rest cause GPFs when trying to register. This is often caused by a corrupted registry.

Registry corruption can be caused by many different things. User interaction and manual editing using REGEDIT can sometimes cause problems. It may also result from corrupt sectors on the hard disk, or other hardware-related error. Under Win 3.11 you also can have the problem with the registry getting corrupted just by writing information using the API commands. One of the main problems that causes corruption with the Win 3.1x registry is the 64K size limit of data contained in the registry. During the first month of shipping OLETools we had a small number of users calling and saying that after they had installed OLETools their registry was corrupted. They discovered this when running VB4 after they had installed OLETools. Their systems displayed a message box telling them that their "Registry is corrupt." Some of these users had backups of their registry before the OLETools install, and sure enough, when we tested the install we found out what was happening. Because of the number of controls and the amount of entries that are added to the registry during installation, an overrun of the 64K limit was occurring and the registry was being corrupted. Unfortunately the API calls for the registry were not returning errors when the registry was full; they were just overwriting parts of the registry at random!

What is even more unfortunate is that there is no way of checking the size of the "internal" data in the registry, and you have no way of knowing (until corruption occurs) whether or not you have hit the magical 64K boundary. The external size (the size reported by the DOS Dir command) varies. Some of the registries contained nearly 64K of data, and their size reported from DOS was 96K. After the corruption occurred it was 59K. We now make a backup copy of the registry, install our controls, and try to read one of the entries from the registry. If we are not able to read our first entry we restore the registry file and let the user know that his or her registry is approaching the 64K point and that he or she might want to clean it up. There are many things you can delete from the registry if they are not being used. For example, VB4 creates OCA files for each OLE control that you use. These files are like "cache" files that let OLE controls load much faster the second and subsequent times that they are used. These files each get an entry in the registry. We have seen many duplications of the OCA files in our registries, so cleaning up these duplicates can provide some much needed space. The only other option you have if your registry is full is to move to a 32-bit operating system, which doesn't have this 64K limitation.

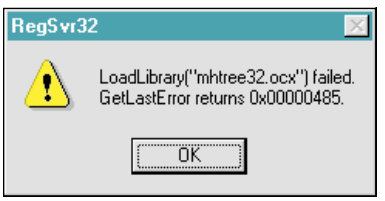

FIGURE 1 ***RegSvr32 Error Message Box.*** *If a file is missing when you try to register, an OLE control error may occur.*

OTHER OCA PROBLEMS

Gustavo Eydelsteyn from VideoSoft wanted to make sure I mentioned a problem that has shown up in VideoSoft's day-to-day testing and in various tech support calls. These problems are caused by OCA files. If you update a control (such as adding new properties, events, or methods), copy that control over an existing control, and then register it, Visual Basic may still behave strangely. Visual Basic will not rebuild the existing OCA file that is there unless you actually delete it *even though the control on which it is based has changed!* If you don't delete it, the incorrect cache information is read from that file. The moral of the story is: make sure you delete any OCA files when installing new controls. Another similar issue is the installation of an upgraded OCX file that has the "OLE" internal version number incremented. For example, if you look in a Visual Basic Project (VBP) file for the reference to the OLE control, it will look something like this:

```
Object={96A6D86D-FE3D-11CE-A0DD-_
   00AA0062530E}#1.0#0; MHCMBO32.OCX
```

This reference loads the control in the project. The line contains a CLSID, the version number of the control, and the file name. When vendors release new major versions of controls they

increment the version number. In the above example, "#1.0#" would change to "#2.0#." The problem lies in the fact that new versions of the control will retain the same name. There will no longer be a version 1.0 control on your system. This confuses Visual Basic and it will say that it can't load your form because the form will look for version 1.0 and the new version of the control is 2.0. To remedy this problem you can update the reference line to point to the new version, 2.0 in this example. Your updated line would look like this:

```
Object={96A6D86D-FE3D-11CE-A0DD-_
   00AA0062530E}#2.0#0; MHCMB032.OCX
```

Another common problem users have encountered follows this scenario: a user installs and registers an OLE control, and then deletes the file, or moves the file to another location and does not reregister it. When you try to insert that object into VB you will get an error, and I have even seen GPFs occur when this happens. A few users had registered controls directly from a CD and then removed the CD, so the controls could not be loaded. This is definitely a user education issue and should start to improve as more OLE-enabled applications are out there. *You must reregister controls if their location is changed!* This same thing applies to any OLE application. When you register an OLE application, it writes information on the location of that server into its registry keys. When OLE tries to create that object, it looks in the registry for the path of the server and tries to start or load that server. If it is no longer present, you get the error.

Another important consideration when installing OLE controls is that you have to reregister a new build of the same control. At first this doesn't seem that important. If you have a new release of the control, and all of its properties, events, methods, and so forth are still the same, why would you have to reregister it? We still haven't figured out why, but if you don't reregister it and just copy it into the same location as the last control, many ugly events may happen. The main one is that often times, VB just won't load the control.

Even though OLE controls are still "new," utilizing some of these hints when distributing your applications or OLE control packages can really help you overcome the issues before they become problems. This information may seem important only to OLE control developers, but it is important to you also. If you use and distribute OLE controls with your application, any one of these problems may happen to you. ×