TECHNICAL FEATURE

Need to extend an unextendible application? It's possible, with Gen Kiyooka's simple, clean, and compatible programming technique.

Extending Windows Executables

Gen Kiyooka

Recently, when creating a commercial software product, I found myself in the unfortunate situation where I needed to extend an application whose source code belonged to someone else, which is to say, the application was not designed to be extensible. While many commercial Windows applications include scripting and macro languages able to load and call arbitrary routines in dynamic link libraries, this particular application was not at that level of sophistication. However, the extensions I had in mind, I felt, more than justified the development cost of trying to figure out how to get inside the application. I began with several possible approaches and fuddled through them before arriving at the solution presented here. Essentially, this article may serve as a guide for anyone wishing to undertake a project of this nature; whether or not the exact solution presented here is made to order will depend on the specifics of your problem.

My solution uses fairly advanced knowledge of several areas of Windows development, notably, the operation of the module loader (dynamic linker), the new executable file format, custom control classes, dynamic link libraries, and Windows instance subclassing. For brevity, some of these techniques will not be explained here, as they are adequately documented elsewhere. Nothing in the following material is specific to any particular language. The caveat, of course, is that the language must support subclassing, the creation of dynamic link libraries, and so on. In practical terms, this includes any of the commercially available C and C++ compilers and Borland International's Pascal products.

Raison d'Etre

While some might consider the practices detailed here to be unsavory or unsafe (or both), I have taken great care in my choice of techniques to ensure that modified Win16 applications should run (in their altered state) under the WOW emulation layer of Windows NT. I also see no reason why these applications would not operate correctly under the Windows emulation layer of OS/2. In short, the techniques below are simply made possible

by exploiting various well-documented but lesser-known (or is that well-known but poorly documented?) aspects of the Windows operating environment. My design goals were to make the alteration simple, clean, and compatible (with any environments where a Win16 application might be asked to execute).

It also had been my intention to ship the altered application as a patch or delta, meaning I would limit the number of actual modifications to the original application. Essentially, I wanted to be able to write a simple patch program that would seek to known offsets in the executable program and perform simple changes to the byte stream, without altering the length or position of information in the .EXE file. However, hexadecimal gymnastics are unnecessary given a fine tool such as RTPatch, in which case, manipulation of arbitrary complexity could be distributed easily via a software patch.

Obviously, you should take care not to violate copyright concerns. My understanding of copyright law is that it is strictly illegal to make a derivative work (alterations included) of a copyrighted piece of software and to distribute that work. The right to create derivative works is one of a bundle of rights comprising the copyright, and it must be granted by the copyright holder in such cases.

Users who purchase copyrighted material are allowed, under the law, to make derivative works for their own personal use. Distributing a patch that alters a copyrighted work for an individual's personal use might be fair game. In any event, consult a good (read: expensive) attorney before modifying and distributing the alterations described here.

A Tale of Two Issues

When I set out to accomplish my task, I uncovered two main issues I felt needed to be addressed to arrive at a robust solution. The first issue was that of getting into the application's process space. Under Windows 3.1, this issue is simplified because all Windows applications execute in a single virtual memory or shared address space. However, under a more robust operating system, like Windows NT, such a feature is not available, and I discarded several possible techniques as a result.

The second issue had to do with taking control or hooking into the user interface of the application. You would expect that adding extensions to an existing application would imply some sort of user interface modification, be it the addition of an existing menu, alterations to a dialogue box, or perhaps sub-classing the behavior of a client window or child window control.

Getting into the Application's Process Space

If you want to add functionality to an existing application, you need to execute code inside that application's address space. Under Windows 3.1, some simplifying assumptions can be made, since all Windows applications execute within a single shared memory address space. However, if you're going to be compatible, you have to imagine that all applications are in disjoint address spaces; therefore, global Win16 techniques will not work. Some of the following ideas I discarded might be useful to a specific operating-system version of an altered program.

System-Wide Message Hook. I discarded the possibility of using a system-wide message hook, as I expected this functionality would not be supported with the same semantics under Win32 running under Windows NT given my experiences with the operating system. I felt it simply would not be possible to build a secure operating system that allows any particular application to just hook into the system-wide message stream. However, the good Dr. GUI, in a recent issue of the *Microsoft Developer Network News*, suggested that system-wide hooks are indeed available in Windows NT.

Under 16-bit Windows, it is possible to use `FindWindow` to locate a particular instance or instance of a particular class. You can then install a global subclass by replacing the Windows procedure in the class structure (say, of the application's frame window) with a substitute window procedure located in a dynamic link library of your own design. Further, an invisible boot-strap application (possibly installed into the `LOAD=` line or the `StartUp` group) could simply create the sub-class and bump up the reference count on the DLL (`LoadLibrary`) and exit, leaving the alteration in place for the remainder of the Windows session. Again, this technique relies on a shared memory address space, and I discarded it to provide for compatibility with Win32 under Windows NT.

TOOLHELP Notifications. A third 16-bit technique might employ the capabilities of the `TOOLHELP` services exposed by the TOOLHELP.DLL. Using `NotifyRegister`, an application can determine when tasks and modules are loaded and freed. Since the goal of these notifications is to allow debuggers to write break points into the address space to be debugged, a clever programmer could simply write some code into a target application's address space due to be executed by the extension.

However, this technique begins to encroach on the notion of a clean alteration. Besides, this is another trick specific to 16-bit Windows and definitely will not work under Win32, at least as far as `TOOLHELP` is concerned. Win32 includes some debugging API calls that provide the same services to debuggers. However, Win32 applications can flag themselves as "not debuggable" under Win32 and Windows NT (security, you must understand), making this technique less than foolproof.

The Holy Grail

The technique that suited my particular problem domain is elegant, reliable, and compatible. It essentially involves modifying an entry in the application's imported name table, a data structure considered to be part of the new executable file format that is documented in a number of places, including the Windows 16-bit SDK documentation.

The imported name table is a list of module names (for most intents and purposes, these are DLL names) that an application is dynamically linked to. The Windows loader uses the imported name table to establish the whereabouts of

DLLs required to resolve references to imported symbols. By using a disk editor (such as the one provided by PC-Tools or Norton Disk Edit) or even Windows Write, you can inspect the imported name table and view the module names of the common imported system DLLs: `USER`, `GDI`, and `KERNEL`.

However, it's likely that any given application imports functions from other modules, some of which likely have only a few exported ordinals. Replacing one of these is definitely the way to go. By altering one or more entries in the imported name table, you can dynamically link an application to a DLL of your own design whenever it executes.

This technique is perfect, as it should work under any compatibility layer intended to run Win16 binaries, and the same thinking can be applied to 32-bit applications that use the portable executable format (PE). The PE format is documented in the Win32 SDK. Ray Duncan in his *PC Magazine* column has done some work uncloaking the mysteries of this beast.

In Your Face

Now that we've established a reliable, clean approach for getting into an arbitrary application's address space at execution time, we need a clean, reliable method of attaching an arbitrary number of adornments to the application's user interface. Again, 16-bit to 32-bit compatibility was of paramount importance to me and led me to discard some easier 16-bit techniques during analysis.

System-Wide or Task-Instanced Message Hook. A system-wide message hook can be a beautiful thing. If you install each available global message hook, you'll be able to filter every message going to every window in the Win16 system. If you know either the Windows class or handle of a particular application, it's easy to filter and generally muck with an application's user interface from inside a message hook.

The downside to this approach is that it can affect performance and it is incompatible with Win32. Ideally, under 16-bit Windows, you might use a hook initially to install a class or instanced subclass on a particular window and then remove the Windows hook shortly thereafter. However, I haven't tried this particular technique and some experimentation is due before I can recommend it wholeheartedly.

A task-filtered hook is another great technique, allowing you to selectively filter the messages for a given task. However, it does not appear to be possible to register a task-specific message hook from inside the `LibMain` of a DLL under Win16, excluding the use of this technique in my particular implementation. Task-instanced hooks should, however, maintain their semantic behavior in the move to Win32, although I haven't tested this theory.

Instance and Class Subclassing

If you know the class name or window text of the particular window you're interested in hooking into, you can obtain (under Win16) a handle to that window using the `FindWindow` API. "Have window handle, will subclass" is the battle-cry of the savvy Windows developer. In my particular case, I replaced the KEYBOARD.DRV entry (`KEYBOARD`) in the imported name table, which meant I was only going to get control of the task during `LibMain`'s execution in my DLL replacement—unfortunately before the host application had registered any window classes or created any windows.

Consequently, this technique was strictly out for me. However, if you end up filtering or thunking a DLL whose entry points get called on a reliable basis after the host application has created its frame window and registered all its window classes, this technique should work just fine. For instance, by replacing KEYBOARD.DRV, you'll need to provide thunks for the ANSI character conversion functions that are repeatedly called to have the keyboard driver perform string translations. From inside your `AnsiToOem` thunk, for instance, you could call `EnumTaskWindows` to walk the Windows list in the current task, subclassing as you saw fit. A complete discussion of subclassing is beyond the scope of this article, although I will discuss a particular implementation.

While many commercial Windows applications can load and call arbitrary routines in dynamic link libraries, some aren't at that level of sophistication.

Choose Me: Modifying Dialogue Box Resources

The technique I chose was to register (or reregister) some child window controls and modify the application's dialogue box resources directly to add control definitions to the dialogue boxes I wanted to hook into. From the outset, I intended to keep things as simple as possible and envisioned patching the executable resource easily. For example, to change the class of a control in a dialogue box template from `Static` to `_tatic`, I would reregister the `Static` class under the new name as a global window class in my DLL.

Obviously, this approach would require writing only a single byte to the executable. For more elaborate interface modifications, I suggest using the Resource Workshop, Whitewater Resource Toolkit, or Microsoft AppStudio to open up the particular dialogue boxes and change them accordingly.

Step By Step: Getting Inside the Process Space

The first step in building your application extension is to reliably get control of execution in the application's address space. The way to do this is with a DLL. But first, we must discuss the behavior of the Windows loader, as implemented in the `KERNEL` module. For the purpose of

brevity and clarity, I will omit many excruciatingly technical details from the description and discuss only the operational model, which is sufficient for our analysis and understanding.

When the user double-clicks an icon in the Program Manager or from another shell program, the **WinExec** function is called, which tells Windows that a program is to be executed. The loader begins by loading the header for the new executable file format into a data structure that has been designated a "module database."

It is the loader's job to resolve external references to the symbols and functions the program requires, but which reside in external DLLs. For instance, **USER**, **GDI**, and **KERNEL** are all DLLs from which every Windows application imports symbols. Obviously, **GDI**, **USER**, and **KERNEL** will always be loaded when the application gets loaded, but what of the other DLLs?

The loader enumerates through the entries in the imported name table and determines if each of the modules listed in the table is currently loaded. The loader attempts to load any unloaded modules before continuing. Essentially, it attempts to load a DLL having the same base name as the module, but with a DLL extension.

For instance, if the named module is called **COMMDLG**, the loader attempts to load the COMMDLG.DLL file. Presumably the loader will find such a DLL in the Windows, Windows system directory, or in the path. If not, the application will not execute, and the system will pop-up a message box telling you that a component required to execute the application is missing.

Exported Ordinals and Identifiers. Once all the modules referenced by name in the imported name table are loaded in the system, the loader fixes up entries in code and data segments that refer to symbols located in these external, dynamically linked modules. There is a table in the new executable format containing relocation information that provides the dynamic linker with the information necessary to do the work.

With respect to function calls, which are of particular interest to us in our quest for a clean alteration, the relocation information takes the form of an index into the imported name table and an ordinal function point identifier. If you're at all familiar with the implementation of a DLL, you'll recognize the ordinal function point identifier as the number given in the module definition file for the DLL. For example:

```
EXPORTS
        Cowabunga       @32
```

The ordinal identifiers for the Windows system DLLs are stabilized over revisions of the system software, allowing applications compiled and linked for Windows 3.0 to work correctly under Windows 3.1, despite the proliferation of new entry points in the 3.1 system DLLs. This fact works in our favor, as we are guaranteed compatibility with future versions of the operating system as long as we stick to documented entry points.

Choose a Suitable DLL We Can "Replace." Using this knowledge of the basic workings of the loader and dynamic linker, we can use Norton Disk Edit (or a similar utility) to fudge the imported name table of an arbitrary application. In my case, I chose the **KEYBOARD** imported name. This implies linking to the KEYBOARD.DRV DLL driver, which exports the language translation functions (like **OemToAnsi**) and provides an interface to the keyboard hardware.

Since this alteration requires us to implement stub functions for each exported ordinal entry point for our chosen module (or DLL), it's a good idea to choose a DLL whose entry points are few and well documented.

In my situation, I simply fudged the name-table entry from **KEYBOARD** to **KEYBOAR_**, another simple modification that can be performed by a simple patch program written in Kernighan and Ritchie C.

Implement a New DLL to Thunk the One We're Replacing. Having changed the imported name from **KEYBOARD** to **KEYBOAR_**, it follows that we will be implementing a DLL called KEYBOAR_.DLL, which exports (at the same ordinal entry points) the functions **OemToAnsi** and **AnsiToOem**. The purpose of these stubs is simply to call through to the original driver using a technique outlined by Charles Petzold in *Programming Windows* and popularized, much to the chagrin of certain legal departments, by Michael Geary in Adobe Type Manager. For readability, and by convention, I chose to implement the functions using name placeholders **xAnsiToOem** and then aliased these functions in the module definition file:

```
EXPORTS
xAnsiToOem    AS    AnsiToOem    @1
  (or whatever)
```

It's critical to make sure the thunk functions are exported with the same ordinal value as their cohorts in the original DLL. Less important is the need to name the function appropriately. A provision exists to have an imported function referenced by name rather than ordinal, but it is in practice rarely used. Completeness is a wonderful thing, however, and should not be undervalued when undertaking program alteration of this nature, especially if your mystical reputation is on the line.

Registering Windows Classes in the Replacement DLL. Now that we've built a new DLL and exported some function

The first step in building your application extension is to reliably get control of execution in the application's address space. The way to do this is with a DLL.

calls for the application to call through, our DLL will be reliably loaded whenever the application is executed. To ensure that the loader can find the DLL under all circumstances, place it in the same directory as the application, in the system directory, or finally, in a directory located in the PATH environment variable. More information about the placement of DLLs can be found in the documentation for the LoadLibrary Windows API.

We'll only be getting control of execution during LibMain and whenever the application calls through any of the function entry points we have thunked. It might be a good idea to register any window classes in LibMain, which is guaranteed to be executed. You can do additional processing in a thunked function once you've determined it will be reliably executed. One reliable method of determining this information is to disassemble the code of the application. Another fine technique is guessing, made more accurate by the Windows APISPY tool developed by Gary Kratkin.

Getting Up in the User Interface

Since subclassing and custom controls are described in a variety of literature, I won't bother repeating the dry details of their implementation. Instead, I'll describe how I've used subclassing to get control of the altered application's user interface.

Modify the Dialogue Box Template. Using Resource Workshop or AppStudio, I opened the executable file for the application already altered once with Norton Disk Edit. Next, I opened up the dialogue box template for the dialogue box to which I intended to add user interface functionality. Finally, I located a static text control located on the surface of the dialogue box and replaced that control with an identical control (that is, having the same numeric identifier and style bits) but with a new class name of _tatic. When the dialogue manager creates the window for this dialogue box template, it will then expect the _tatic class to be registered in the system.

Reregister the Static Class Under a New Name. Next, in the LibMain for my DLL, I reregistered the class Static under the new name _tatic, giving a new WndProc that I own. Essentially, I've now ensured that I will get control of execution when that dialogue box is displayed. When the dialogue box is created, the dialogue manager will create an instance of the _tatic class that my DLL owns, allowing me to get execution in the window procedure of the _tatic class.

The underlying application, of course, has no idea that any of this whiz-bangery is taking place.

Subclass the Parent of the _tatic Class in WM_CREATE. Finally, I need yet another subclass (YASC). In this case, it's an instanced subclass to be performed on the parent (that is, the hDlg) of the child window control during the processing of the child window's WM_CREATE message. This involves fetching the window handle to the parent using the GetParent API and employing SetWindowLong to substitute a pointer to the new-and-improved Windows procedure into the parent's window structure.

As this Windows procedure is essentially the DlgProc of the dialogue box, I now have complete access to the message stream for that dialogue box. To that end, I can process the WM_INITDIALOG message in the subclass procedure (of the parent dialogue) and go ahead and add any buttons to the dialogue or change its behavior. The underlying application, of course, has no idea that any such whiz-bangery is taking place.

Altered States

Obviously, this technique opens up a virtual Pandora's box of possibilities, both beneficial and malicious. Be prudent during your software alteration and test like crazy before relying on these techniques in commercial software. ■

Gen Kiyooka recently finished developing RoboHelp 2.0 and is undergoing postnatal therapy.