Writing Your Own Install and Uninstall Code

With a self-deleting DLL you can create a self-deleting executable

WHEN I WROTE MY Windows program What To Do, I knew I was going write my own install and uninstall code. I wanted to have full control over what users saw from the moment they began their install to the last thing they saw during a possible uninstall. This article covers some of the techniques I found most useful.

Auto Uninstall

The most challenging part of writing the uninstaller was figuring out how to make it delete itself after it was done removing program files and directories. I wanted it to work on everything from Windows 95 to Windows XP, without making the user download any additional components. I searched on the Web and found several references to self-deleting executables, but all the proposed solutions had problems. Most only worked with certain versions of Windows. Others modified thread priorities in ways that could cause timing problems. Some caused extra windows or error messages to appear. I figured there must be a better solution. I discovered that you can use a self-deleting DLL to create a self-deleting executable with none of the limitations of previous solutions.

The Windows rundll32.exe Utility

How can we execute code in a DLL without creating an EXE to load and call it? Every Windows version beginning with Windows 95 has shipped with a system utility called "rundll32.exe." It allows you to execute any function exported from a DLL. You use it like this:

```
rundll32.exe DllName,ExportedName args
```

`ExportedName` is the exported name of the function in the DLL. When writing a DLL to use with rundll32, I declare the function to be exported like this:

```
extern "C" __declspec(dllexport)
void CALLBACK FunctionName (
  HWND hwnd,
  HINSTANCE hInstance,
  LPTSTR lpCmdLine,
  int nCmdShow
)
{ … }
```

The rundll32.exe documentation lists the function arguments, but empirically, I've found that the only value you can rely on is `lpCmdLine`, which receives the value of args passed when you run rundll32.exe. Using `__declspec(dllexport)` causes the function to be exported, and using `extern "C"` makes the exported name `_FunctionName@16` (the function name gets mangled to include the size of the function arguments).

rundll32.exe loads the specified DLL and then calls the exported function passing as args the value of `lpCmdLine`. The official documentation for rundll32.exe can be found at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tools/tools/rundll32.asp.

A Self-Deleting DLL

Listing 1 shows the source code for a DLL that will delete a file and then delete itself. `DllMain` will be called when the DLL is first loaded, and there it records the module handle, which will be used later to get the DLL's filename.

In the function `MagicDel`, `lpCmdLine` will be the name of the executable file that the DLL should delete (for example, the uninstaller's filename). Deleting it is easy—sleep for a while to allow the executable's process time to exit and then call `DeleteFile`. To get fancier, pass the handle of the executable's process to `MagicDel` and wait on it before calling `DeleteFile`.

Making the DLL delete itself is a little trickier. rundll32 calls `LoadModule` to load the DLL into its address space. If the DLL function was allowed to return, rundll32 would just exit, causing the DLL to be released (but not deleted). Instead of letting that happen, we want to execute the following code:

```
FreeLibrary(DLL module handle);
DeleteFile(DLL filename);
ExitProcess(0);
```

The function can't just make this sequence of calls directly because `FreeLibary` would make the code page invalid. To work around this, the function pushes an equivalent set of assembler instructions onto the stack and then

ALEX TILLES *is the creator of What To Do, a to-do list manager that adapts to the way you organize. He was also a development lead at Microsoft for several years in the Systems and Internal Tools groups. He can be reached at alex@handcraftedbytes.com.*

Listing 1
Source for MagicDel.dll—a self-deleting DLL

```
#include <windows.h>
HMODULE g_hmodDLL;

extern "C" BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD reason, LPVOID)
{
    if (reason == DLL_PROCESS_ATTACH)
        g_hmodDLL = hinstDLL;
    return TRUE;
}

extern "C" __declspec(dllexport) void CALLBACK MagicDel(HWND,
                                                        HINSTANCE,
                                                        LPTSTR lpCmdLine,
                                                        int)
{
    // delete the executable file that created this process
    Sleep(2000);
    DeleteFile(lpCmdLine);

    // delete ourself
    char filenameDLL[MAX_PATH];
    GetModuleFileName(g_hmodDLL, filenameDLL, sizeof(filenameDLL));

    __asm
    {
        lea     eax, filenameDLL
        push    0
        push    0
        push    eax
        push    ExitProcess
        push    g_hmodDLL
        push    DeleteFile
        push    FreeLibrary
        ret
    }
}
```

begins executing them with a `ret` instruction. The final call to `ExitProcess` prevents the process from trying to run any more of the code. I created the assembly code block using the same technique that Gary Nebbit presented in a former *Windows Developer Network* "Tech Tip" (see http://www.windevnet.com/documents/s=7257/wdj0109g/).

If you build the DLL using Visual Studio with default options, the resulting binary is about 40K. By removing the unused C runtime code, you can shrink it to 2.5K (see the sidebar "Shrinking the DLL").

A Self-Deleting Executable

An executable can delete itself by storing a copy of the self-deleting DLL as a resource, then recreating it and launching a process running rundll32.exe to do the deletion.

Listing 2 shows the header and resource files used to store the DLL as a resource. Resource type values 256 and above are available for user-defined types. Alternately, the DLL binary could be stored directly in source as an array of bytes.

Listing 3 shows the rest of the code for the executable. `WriteResourceToFile` accesses the binary resource so that it can recreate the DLL on disk. The Windows Resource APIs provide a pointer to the raw data.

`SelfDelete` recreates the DLL and then builds the command line to launch rundll32.exe. The command line will look like this:

```
path\rundll32.exe magicdel.dll,_MagicDel@16 path\executableName
```

rundll32.exe resides in either the Windows or System directory, so `SelfDelete` tests to find the right location. When `CreateProcess` is called to execute the command line, it sets the STARTF_FORCEOFFFEEDBACK flag to prevent Windows from displaying the busy cursor while rundll32.exe runs. That way, there will be no indication to the user that a new process is running. After this new process exits, both the DLL and the original executable are gone.

To make a self-deleting executable that does not depend on the C runtime library DLL, the executable must statically link in the runtime library code. Change the project options under C/C++/Code Generation to set "Runtime Library" to either "Single Threaded" or "Multi Threaded" (or any value that does not include the DLL).

This self-deleting technique works reliably with all Windows versions. The actual uninstaller for my Windows program first copies itself to the Windows temp directory so that it can remove all program files and directories. Finally, it uses the self-deleting DLL to delete itself.

Shrinking the DLL

THE SELF-DELETING DLL doesn't use any C runtime functions, so you can make it a lot smaller by removing the C runtime library code. Using Visual Studio .NET to build the DLL, change the DLL project options under Linker/Input to set "Ignore All Default Libraries" to Yes. This passes /NODEFAULTLIB to the linker and so prevents the inclusion of the runtime code. Since the runtime library normally provides the entry point for the DLL, you have to explicitly change project options under Linker/Advanced to set "Entry Point" to `DllMain`.

If you now try to build the DLL, you'll get two unresolved externals, `__security_cookie` and `@__security_check_cookie@4`. Change the project options under C/C++/Code Generation to set "Buffer Security Check" to No. This prevents the /GS flag from being passed to the compiler and gets rid of the unresolved externals.

The resulting DLL is just 2.5K.

—A.T.

Install

I knew how I wanted my installation program to work. The user would just download my executable over the Internet and run it. I wanted the file to be compressed for faster downloading, and I wanted the first thing the user saw to be my program instead of some other company's installer. Fortunately, Windows provided just the support I needed.

First, I created an interactive setup program that displays a license agreement, prompts the user for install choices, copies files, and does the rest of the setup work. Then I stored a compressed version of that setup program as a binary resource inside the installer. All the installer has to do is write the binary resource to disk, decompress it, and launch it as a new process. Storing and writing the binary resource was easy—I used the aforementioned binary resource handling code.

Every Windows platform since Windows 95 has had an API for decompressing files—`LZCopy`. Listing 4 shows the source for the installer that makes use of it. The compressed setup program is stored as a binary resource, just as before. `DecompressFile` shows how to use the

Listing 2
Including a file as a resource

```
//  SelfDelete.h
#define RC_BINARYTYPE 256
#define ID_MAGICDEL_DLL 100


//  SelfDelete.rc
#include "SelfDelete.h"
ID_MAGICDEL_DLL RC_BINARYTYPE MagicDel.dll
```

`LZCopy` API. The installer recreates the AppSetup.exe, and then runs it. To build successfully, the installer executable needs to add lz32.lib to the project options under Linker/Input/Additional Dependencies. And as before, statically link in the runtime library code by changing the project options under C/C++/Code Generation to set "Runtime Library" to either "Single Threaded" or "Multi Threaded." Note that the installer doesn't have to wait for the setup program to complete because AppSetup.exe can use the self-deleting DLL to delete itself when it's done.

The trickiest part of using `LZCopy` is that it seems to only decompress files that have been compressed with the Microsoft compression utility compress.exe, a command-line utility that used to ship in some Microsoft SDKs. The only official source for it now appears to be as part of a self-extracting archive available at ftp://ftp.microsoft.com/softlib/mslfiles/CP0982.EXE. If you download and run that, it will unpack compress.exe (as well as several other files that you can ignore and delete). You use it like this:

```
compress SourceName DestinationName
```

Using the decompression support built into all versions of Windows made it easy to write the installer. Note that all Windows versions include the utility expand.exe, which allows you to decompress files from the command line.

Complete Control

Using a self-deleting DLL, binary resources, and the decompression support built into Windows can help you create your own installer and uninstaller. And that allows you to control every facet of the experience from the moment users begin installing your software. **w::d**

| Download code › **windevnet.com/wdn/code/** |

Listing 3
A self-deleting executable

```
#include <windows.h>
#include "SelfDelete.h"

void WriteResourceToFile(HINSTANCE hInstance,
                         int idResource,
                         char const *filename)
{
    //  access the binary resource
    HRSRC hResInfo = FindResource(hInstance, MAKEINTRESOURCE(idResource),
                                  MAKEINTRESOURCE(RC_BINARYTYPE));
    HGLOBAL hgRes = LoadResource(hInstance, hResInfo);
    void *pvRes = LockResource(hgRes);
    DWORD cbRes = SizeofResource(hInstance, hResInfo);

    //  write the binary resource to a file
    HANDLE hFile = CreateFile(filename, GENERIC_WRITE, 0, 0, CREATE_ALWAYS,
                              FILE_ATTRIBUTE_NORMAL, 0);
    DWORD cbWritten;
    WriteFile(hFile, pvRes, cbRes, &cbWritten, 0);
    CloseHandle(hFile);
}

void SelfDelete(HINSTANCE hInstance)
{
    WriteResourceToFile(hInstance, ID_MAGICDEL_DLL, "magicdel.dll");

    //  Build command line
    //  1. Find rundll32.exe
    char commandLine[MAX_PATH * 3];
    GetWindowsDirectory(commandLine, sizeof(commandLine));
    lstrcat(commandLine, "\\rundll32.exe");

    if (GetFileAttributes(commandLine) == INVALID_FILE_ATTRIBUTES)
    {
        GetSystemDirectory(commandLine, sizeof(commandLine));
        lstrcat(commandLine, "\\rundll32.exe");
    }

    //  2. Add rundll32.exe parameters
    lstrcat(commandLine, " magicdel.dll,_MagicDel@16 ");

    //  3. Add this file name
    char thisName[MAX_PATH];
    GetModuleFileName(hInstance, thisName, sizeof(thisName));
    lstrcat(commandLine, thisName);

    //  Execute the command line
    PROCESS_INFORMATION procInfo;
    STARTUPINFO startInfo;
    memset(&startInfo, 0, sizeof(startInfo));
    startInfo.dwFlags = STARTF_FORCEOFFFEEDBACK;
    CreateProcess(0, commandLine, 0, 0, FALSE, NORMAL_PRIORITY_CLASS, 0, 0,
                  &startInfo, &procInfo);
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    SelfDelete(hInstance);
}
```

Listing 4
A decompressing installer

```
//  install.h
//
#define RC_BINARYTYPE           256
#define ID_COMPRESSED_SETUP     100

//
//  install.rc
//
#include "install.h"
ID_COMPRESSED_SETUP     RC_BINARYTYPE    AppSetup.ex_


//
//  install.cpp
//
#include <windows.h>
#include "install.h"

void WriteResourceToFile(HINSTANCE hInstance,
                         int idResource,
                         char const *filename)
{
    //  see Listing 3
}

void DecompressFile(char const *source, char const *dest)
{
    OFSTRUCT ofs;
    ofs.cBytes = sizeof(ofs);
    int zhfSource = LZOpenFile(const_cast<char *>(source), &ofs, OF_READ);
    int zhfDest = LZOpenFile(const_cast<char *>(dest), &ofs,
                  OF_CREATE | OF_WRITE);
    LZCopy(zhfSource, zhfDest);
    LZClose(zhfSource);
    LZClose(zhfDest);
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    WriteResourceToFile(hInstance, ID_COMPRESSED_SETUP, "AppSetup.ex_");
    DecompressFile("AppSetup.ex_", "AppSetup.exe");
    DeleteFile("AppSetup.ex_");

    //  launch AppSetup.exe
    PROCESS_INFORMATION procInfo;
    STARTUPINFO startInfo;
    memset(&startInfo, 0, sizeof(startInfo));
    CreateProcess(0, "AppSetup.exe", 0, 0, FALSE, NORMAL_PRIORITY_CLASS, 0, 0,
                  &startInfo, &procInfo);
}
```