How were DLL functions exported in 16-bit Windows?

devblogs.microsoft.com/oldnewthing/20060714-16

July 14, 2006

Raymond Chen

The whole point of dynamic link libraries (DLLs) is that the linkage is dynamic. Whereas statically-linked libraries are built into the final product, a module that uses a dynamically-linked library merely says, “I would like function X from Y.DLL, please.” This technique has advantages and disadvantages. One advantage is more efficient use of storage, since there is only one copy of Y.DLL in memory rather than a separate copy bound into each module. Another advantage is that an update to Y.DLL can be made without having to re-compile all the programs that used it. On the other hand, the ability to swap in functionality automatically is also one of the main disadvantages of dynamic link libraries, because one program can change a DLL that has cascade effects on other clients of that DLL. Anyway, let’s start with how 16-bit Windows managed imports and exports. After that, we’ll see how things changed during the switch to 32-bit Windows, and then we’ll take a look at the compiler-specific `dllimport` declaration specifier. (I already discussed `dllexport` earlier.) A 16-bit DLL has not one but three export tables. (Things are actually more complicated than I describe them here, but I’m going to skip over the nitpicky details just to keep everyone’s heads from exploding.) The most important table is a sparse array of functions, indexed by a 1-based integer (the “ordinal”). It is this function table that is the master list of all exported functions. If you request a function by ordinal, the ordinal is looked up in this table. The table is physically rather complicated due to the sparseness, but logically, it looks like this:

Ordinal	Address	other goo
1	02:0014	…
2	04:0000	…
5	02:02C8	…

The first column in the table is the ordinal of the function, and the second function describes where the function can be found. (Notice that there is no function 3 or 4 in this DLL.) Things get interesting when you want to export a function by name. The exported names table is a list of function names with their associated ordinal equivalents. For example, a section of the exported names table for the 16-bit window manager (USER) went like this:

…

`ClipCursor`	16
`GetCursorPos`	17
`SetCapture`	18

…

If somebody asks for the address of the function `ClipCursor`, the exported names table is consulted, the value 16 is retrieved, and the function at position 16 in the ordinal export table is returned. Although you can't see it here, there was no requirement that the names in the exported names table be in any particular order, or that every ordinal have a corresponding name. Wait, did I say the exported names table? I'm sorry, that was an oversimplification. There are actually two exported names tables, the resident names table and the non-resident names table. As their names suggest, the names in the resident names table remain in memory as long as the DLL is loaded, whereas the names in the non-resident names table are loaded into memory only when somebody calls `GetProcAddress` (or one of its moral equivalents). This distinction is a reflection of the extremely tight memory constraints that Windows had to run within back in those days. For example, the window manager (USER) has over six hundred export functions; if all the exported names were kept resident, that would be over ten kilobytes of data. You'd be wasting four percent of the memory of your 256KB machine remembering things you don't need most of the time. The large size of the table for exported function names meant that only functions that are passed to `GetProcAddress` with high frequency deserve to be placed in the resident names table. For most DLLs, no function falls into this category, and the resident names table is empty. (Head-exploding details deleted for sanity's sake.) Since obtaining a function by name is so expensive (requiring the non-resident names table to be loaded from disk so it can be searched), all functions exported by operating system DLLs are exported both by name and by ordinal, with the ordinal taking precedence in the import library table. Obtaining a procedure address by ordinal avoids the name tables entirely. Notice that every named function has a corresponding ordinal. If you do not assign an ordinal to your named function in your module definition file, the linker will make one up for you. (However, the value that it makes up need not be the same from build to build.) This situation did not occur in practice, for as we noted above, everybody explicitly assigned an ordinal to their exports and put that ordinal in the import library in order to avoid the huge cost of a name-based function lookup.

That's a quick look at how functions were exported in 16-bit Windows. Next time, we'll look at how they are imported.

Raymond Chen

Follow