How do I convert a method name to a method index for the purpose of INTERFACEINFO?

devblogs.microsoft.com/oldnewthing/20130329-00

March 29, 2013

Raymond Chen

The `IMessageFilter::HandleIncomingCall` method describes the incoming call by means of an `INTERFACEINFO` structure:

```
typedef struct tagINTERFACEINFO {
  LPUNKNOWN pUnk;
  IID iid;
  WORD wMethod;
} INTERFACEINFO, *LPINTERFACEINFO;
```

The `wMethod` is a zero-based index of the method within the interface. For example, `IUnknown::QueryInterface` has index zero, `IUnknown::AddRef` has index one, and `IUnknown::Release` has index two.

If you want to filter on a method in an interface, you need to know its index. One way of doing this would be to sit and count the methods, but this is error-prone, especially if the interface is still under active development and is not yet set in stone.

C to the rescue.

The IDL compiler spits out a C-compatible structure for the virtual function table, and you can use that structure to derive the method indices. For example:

```
#if defined(__cplusplus) && !defined(CINTERFACE)
    …
#else   /* C style interface */
    typedef struct IPersistStreamVtbl
    {
        BEGIN_INTERFACE


        HRESULT ( STDMETHODCALLTYPE *QueryInterface )(
            __RPC__in IPersistStream * This,
            /* [in] */ __RPC__in REFIID riid,
            /* [annotation][iid_is][out] */
            _COM_Outptr_  void **ppvObject);


        ULONG ( STDMETHODCALLTYPE *AddRef )(
            __RPC__in IPersistStream * This);


        ULONG ( STDMETHODCALLTYPE *Release )(
            __RPC__in IPersistStream * This);


        HRESULT ( STDMETHODCALLTYPE *GetClassID )(
            __RPC__in IPersistStream * This,
            /* [out] */ __RPC__out CLSID *pClassID);


        HRESULT ( STDMETHODCALLTYPE *IsDirty )(
            __RPC__in IPersistStream * This);


        HRESULT ( STDMETHODCALLTYPE *Load )(
            __RPC__in IPersistStream * This,
            /* [unique][in] */ __RPC__in_opt IStream *pStm);


        HRESULT ( STDMETHODCALLTYPE *Save )(
            __RPC__in IPersistStream * This,
            /* [unique][in] */ __RPC__in_opt IStream *pStm,
            /* [in] */ BOOL fClearDirty);


        HRESULT ( STDMETHODCALLTYPE *GetSizeMax )(
            __RPC__in IPersistStream * This,
            /* [out] */ __RPC__out ULARGE_INTEGER *pcbSize);


        END_INTERFACE
    } IPersistStreamVtbl;
    …
#endif  /* C style interface */
```

(You get roughly the same thing if you use the `DECLARE_INTERFACE` macros.)

After we remove the distractions, the structure is just

```
typedef struct IPersistStreamVtbl
{
    BEGIN_INTERFACE
    HRESULT (*QueryInterface)(…);
    ULONG (*AddRef)(…);
    ULONG (*Release)(…);
    HRESULT (*GetClassID)(…);
    HRESULT (*IsDirty)(…);
    HRESULT (*Load)(…);
    HRESULT (*Save)(…);
    HRESULT (*GetSizeMax)(…);
    END_INTERFACE
} IPersistStreamVtbl;
```

From this, we can write a macro which extracts the method index:

```
// If your compiler supports offsetof, then you can use that
// instead of FIELD_OFFSET.
#define METHOD_OFFSET(itf, method) FIELD_OFFSET(itf##Vtbl, method)


#define METHOD_INDEX(itf, method) \
    ((METHOD_OFFSET(itf, method) - \
      METHOD_OFFSET(itf, QueryInterface)) / sizeof(FARPROC))
```

The macro works by looking at the position of the method in the vtable and calculating its index relative to `QueryInterface`, which we know has index zero for all `IUnknown`-derived COM interfaces.

These macros assume that the size of a pointer-to-function is the same regardless of the prototype, but this assumption is safe to make because it is required by the COM ABI.

Observe that in order to get the C-style interfaces, you must define the `CINTERFACE` macro before including the header file. (And observe that the C-style interfaces are not available in C++; you must do this in C.)

If the bulk of your program is in C++, you can slip in a single C file to extract the method indices and expose them to the C++ side either through global variables or short functions. Depending on how fancy your link-time code generator is, the global variable or function call might even become eliminated.

Raymond Chen

Follow