Rudy's Delphi Corner

Pitfalls of converting

Translation is the art of failure. — Umberto Eco

This article is meant for everyone who needs to translate C/C++ headers to Delphi. I want to share some of the pitfalls you can encounter when converting from C or C++. This article is not a tutorial, just a discussion of frequently encountered problem cases. It is meant for the beginner as well as for the more experienced translator of C and C++.

Notes

I never use notes, they interfere with me. — Ken Blanchard

This is not a tutorial on header conversion. You can find a tutorial on the JEDI site. You can also start on the JEDI API how-to page. This article assumes you have sufficient knowledge of the Delphi language, and some (reading) knowledge of C and/or C++. You should also have read the documents on conversion, formerly on the JEDI pages, and the Borland conversion documentation, if you want your conversion to be compliant with Borland or JEDI standards (this is not required though).

I have found another tutorial for the translation of C and C++ (not only headers) to Delphi. I didn't read it in full, so I don't know how accurate it is. But it seems to discuss every construct.

In this article I often use the word "function", since that is how they are called in C and C++. In Delphi, you have both functions and procedures, but you might say that procedures are just a special case of functions, without a return value. So if I write "function", I mean both "function" and "procedure". Sometimes I use the word "routine" as well, with the same meaning.

When the conversion must also be usable in C++Builder, you must take care not to forget to use the `{$EXTERNALSYM SomeSymbol}` and perhaps the `{$NODEFINE SomeSymbol}` directives where appropriate. More on this later on. In the sources displayed in the article, I purposely omitted most of them, because they greatly decrease the signal-to-noise ratio.

C and C++ use a lot of pointers, in structures, in parameters, etc. If you have problems understanding or using pointers, you should perhaps read my article "Addressing pointers".

To help me with some of the most annoying tasks of header conversion, I wrote a simple IDE expert for myself. This is also available for download.

Not for .NET

This article is (currently) entirely meant for the Win32 (and Win64) versions of Delphi. The principles explained here don't fully apply to the Microsoft .NET versions, using *PInvoke*, although some parts do. People using .NET might be interested in using the .NET translations of Win32 functions on PINVOKE.NET. Since this site produces C# translations, this can be combined (if you want, since you can also use the C# functions directly) with Embarcadero's Babelcode webservice.

Calling conventions

True art selects and paraphrases, but seldom gives a verbatim translation. — Thomas Bailey Aldrich

Probably the easiest, and most often made mistake is using the wrong calling convention. Now you may say: "My code is unconventional, so I don't need them anyway", but it's not that simple. So what is a calling convention?

To call a function, you must often pass parameters to it. There are plenty ways how this can be done. In old BASIC programs, you would just load some global variables with values and use these in your subroutine. Fortunately, things have become a bit more sophisticated, and we know two major ways of doing it (on Windows). You either pass the parameters on the calling stack (the place the processor also places the temporary pointer to the code following the call, so it knows where to continue after the call was done), or you pass some of them in registers. Floating point values can also be passed on the stack of the coprocessor.

Calling conventions rule how parameters will be passed (stack only or registers), in which order they will be passed (from left to right, i.e. in the same order as they appear in source code, or the other way around), and which code will clean the stack after use, if necessary. There are a lot of possible combinations, but Delphi currently knows 5 (or well, 6, if you count *winapi*) of them:

- *pascal*, the original but mostly obsolete calling convention for old Pascal programs;
- *register*, the current default calling convention in Delphi (*__fastcall* in C++Builder);
- *cdecl*, the standard calling convention for C and C++ code;
- *stdcall*, the default cross-language calling convention on 32-bit Windows;
- *winapi*, an *undocumented* but supported alias for *stdcall*; and
- *safecall*, a special case of *stdcall*, which can be ignored for now.

The problem that often arises is, that either no calling convention is mentioned at all, or the wrong one is used. If you don't mention any convention in your code, this means the Delphi compiler will use *register*. But if the original C/C++ code didn't mention one either, it will use *cdecl* as a standard. So your converted code will not work, or worse, it will cause a crash.

There are a few ways to try to determine the calling convention. First, C/C++ either uses *__cdecl* (by default), but can be instructed to use *__stdcall* or *__pascal*, or even *__fastcall* (register). The latter is however not completely compatible with Delphi's *register* (different registers are being used). To make things even more confusing, C++ Builder also has a *__fastcall*, which is compatible with Delphi, but also knows the *__msfastcall* convention, which is compatible with Microsoft's *__fastcall*. We may see this in a newer version of Delphi too. But if you see these keywords, you know which convention is being used.

Note: the usual calling convention for Windows DLLs is stdcall. Most of the Windows APIs use this convention. In Linux, there is no stdcall, and the default is cdecl there.

Macros

But the convention can be hidden in a macro (often an all capital word, like *PASCAL*). These are often #defined in another header file, to which you many not even have access. If you do, follow all includes until you have found one of the keywords mentioned above (i.e. *_pascal* or *__stdcall*, don't mind the number of underscores, that's a C compiler thing). Be aware that the values of the macros can be different for different platforms, so make sure the type you found is the definition for Win32, and not for the Mac or the Alpha.

One case that is really annoying is `FAR PASCAL`. You might expect this to be translated to `far; pascal;` but this is far from the truth. In 32-bit Windows, *PASCAL* seems to be #defined as *__stdcall*, and *FAR* as an empty macro (*far* is something that was necessary for segmented 16-bit DOS/Windows 3.x code, not for Windows 32-bit code). So to find out the correct calling convention may require some detective work. A good searching tool, like grep, can help you with this.

The most common macros are these:

Macro	C++ definition	Delphi definition
CALLBACK	__stdcall	stdcall
WINAPI	__stdcall	stdcall
WINAPIV	__cdecl	cdecl
APIENTRY	WINAPI	stdcall
APIPRIVATE	__stdcall	stdcall
PASCAL	__stdcall	stdcall
FAR	*(nothing)*	*(nothing)*

Other macros are either defined in terms of these macros, or directly as *__stdcall*, *__cdecl*, etc.

A trick

If you can't find the place where the macro was defined, there is one trick left. Normally, under 32-bit Windows, you can expect *stdcall*. However, some headers still use *cdecl*. To find out which one it is, you should do a test. Declare only one of the functions (but one with parameters) in the header as *stdcall*. Now call this function in code, and put a breakpoint on it. Debug the code, and as soon as the breakpoint is reached, open the CPU window of the IDE. Note the value of the *ESP* register. Now step over the function with **F8**, and note the *ESP* value again. If it was changed, you were right, and it was a *stdcall* function (the function cleaned the stack, changing the stack pointer as an effect). If it didn’t change, it was a *cdecl* function. If you are familiar with assembler, you might even directly see the cleaning of the stack (something like `ADD ESP,xxx`).

Win64

In 64 bit Windows (for AMD or Intel), there is only one public calling convention. It is described on the MSDN site. DLLs will and should not use anything else. Delphi can use and produce it.

If you want to share your translations between Win32 and Win64, there is no need to remove declared calling conventions like *stdcall* or *cdecl*. The Win64 compiler will simply ignore them.

Parameters

> *If you have a procedure with 10 parameters, you probably missed some.* — Alan Perlis

Parameter passing in C or C++, and Delphi is often different. In C, all parameter passing is “by value”, which means that a copy of the value is passed to the routine. In C++, there is also passing by reference, when a reference type is used (like `int& myInt`). In headers, you’ll hardly ever see the latter. Delphi is a little richer however, and even more is Microsoft’s MIDL (Microsoft Interface Definition Language). I assume that you are familiar with the data types, and the parameter declarations in C and C++, since I don’t want to discuss them. I only want to point out, where something that seems obvious could go wrong, because of the differences in language.

Const parameters

You’ll often see the *const* keyword being used. For parameters, *const* in Delphi has quite a different meaning than in C or C++. In Delphi, it means that the parameter will not be changed in the routine. This is important for reference counted types, like strings, dynamic arrays, variants and interfaces, and for “large” types. For large types, it only has importance for the Delphi code that is being generated, not for the interface of the function. For reference counted types, it can be important.

Since Delphi strings and dynamic arrays should not be passed as reference counted types to API functions anyway (more about this later on), this only leaves interface types.

First a bit about const parameters in Delphi. They will either be passed by value, or by reference, depending on which of these is more efficient. Since the rules are not too easy to remember, and different from what *const* means in C, the safest thing you can normally do, is to ignore the C *const* keyword alltogether. It will make no difference to your code, and not to the code being called either.

Interfaces are different. However, it is best not to declare them as *const* either. Omitting *const* will cause a call to *_AddRef* before the call, and one to *_Release* afterwards, but that doesn’t affect either the calling code, nor the code being called. It is usually safer for the lifetime of the interface.

Pointer parameters

Pointer parameters are often hard to translate properly. Embarcadero’s documents tell you to use *var* wherever possible, to make the code more Delphi-like. “Wherever possible” means, that pointers should be converted as *var* parameters (i.e. passed by reference), if you can be sure that never a *nil* value can be expected, and that the pointer is not pointing to an array (more on this later). This requires a thorough study of the API documentation, since this information is not always very obvious. If you are in doubt, don’t use *var*, but translate to a pointer type, even if this means that you’ll have to define it. If you do so, you should preferrably define the pointer type with a `{$NODEFINE}` directive, so it will not appear in a .hpp file for C++Builder.

An example:

```
virtual HRESULT STDMETHODCALLTYPE GetBindString(
            /* [in] */ ULONG ulStringType,
            /* [out][in] */ LPOLESTR __RPC_FAR *ppwzStr,
            /* [in] */ ULONG cEl,
            /* [out][in] */ ULONG __RPC_FAR *pcElFetched) = 0;
```

If we remove the – for us – unnecessary macros and comments:

```
virtual HRESULT STDMETHODCALLTYPE GetBindString(
            ULONG ulStringType,
            LPOLESTR *ppwzStr,
            ULONG cEl,
            ULONG *pcElFetched) = 0;
```

this will become (documentation said the pointer points to an array):

```
type
  {$NODEFINE POLEStrArray}
  POLEStrArray = ^TOLEStrArray;

  {$NODEFINE TOLEStrArray}
  TOLEStrArray = array[0..High(Integer) div SizeOf(POLEStr) - 1 of POLEStr;

function GetBindString(ulStringType: ULONG;
   wzStr: POLEStrArray; cEl: ULONG;
   var pcElFetched: ULONG): HResult; stdcall;
```

Incidentally, I do not fully agree with Borland's and now Embarcadero's argumentation. If you use an API, the documentation for it will probably use C syntax. Using *var* most of the time, but not always, will require the user of the translation to look up the actual translation in the source (although the IDE will give you hints, if you use that feature). Therefore I'd advocate the use of a stricter translation, which is as close to the original as possible, even if this means that you don't use *var* at all, but only pointer types instead. You could also use different, overloaded versions, and provide both translations, but that has also a few pitfalls of its own (more about that later on).

I guess the decision to use *var* where possible was made a long time ago, and changing it now would probably break too much existing code around.

Array parameters

Array parameters are hard to convert. The semantics for pointers and arrays are quite a bit different in C/C++ and Delphi. In C/C++, an array variable is no more than a pointer to its first element, so pointer and array types can often be mixed as wished. C and C++ don't have built-in range checks, so this can be quite dangerous.

Note that I am not saying that arrays and pointers are the same thing in C and C++. They are not. For instance, in these languages, you can assign an array to a pointer variable, but you can't do the opposite. Otherwise, the variables can often be used as if they were the same. The real difference is their initialization and storage, but for this article, that is unimportant.

So how do you convert the following?

```
WINCRYPT32API BOOL WINAPI CertVerifyCRLRevocation(
    IN DWORD dwCertEncodingType,
    IN PCERT_INFO pCertId,
    IN DWORD cCrlInfo,
    IN PCRL_INFO rgpCrlInfo[]
);
```

That could become:

```
type
  PPCRL_INFO = ^PCRL_INFO;

function CertVerifyCRLRevocation(
  dwCertEncodingType: DWORD; pCertId: PCERT_INFO;
  cCrlInfo: DWORD; rgpCrlInfo: PPCRL_INFO): BOOL; stdcall;
```

In other words, you can turn an array like that into a pointer. This way you can iterate over the array with simple pointer arithmetic like `Inc(MyPtr);` and you don't have to declare a complicated type. A better approach would however be:

```
type
  {$NODEFINE PPCRL_INFOArray}
  PPCRL_INFOArray = ^TPCRL_INFOArray;

```

```
    {$NODEFINE TPCRL_INFOArray}
    TPCRL_INFOArray = array[0..65535] of PCRL_INFO;

  function CertVerifyCRLRevocation(
    dwCertEncodingType: DWORD; pCertId: PCERT_INFO;
    cCrlInfo: DWORD; rgpCrlInfo: PPCRL_INFOArray): BOOL; stdcall;
```

This way the array can really be used as an array of *PCRL_INFO*. I used 65535 here, since often that is a suitably high value. If not, you can define the array as:

```
TPCRL_INFOArray = array[0..High(Integer) div
                    SizeOf(PCRL_INFO) - 1] of PCRL_INFO;
```

But usually, this is not necessary. And if you do things like these, be sure not to come too close to `High(Integer)`, since the compiler might complain that the data structure would be too large.

On another note, a C user might say that the *PPCRL_INFOArray* is one indirection too many. For a C programmer it would be, but not for Delphi. In C, the array is a pointer to the first element. In Delphi, the array is not a pointer type, so you'll have to use an extra pointer type, not the straight array declaration.

Now you would perhaps be tempted to use this declaration of the *rgpCrlInfo* parameter:

```
var rgpCrlInfo: array of PCRL_INFO;
```

This would indeed be a very nice syntax, but it is not the same as the above array definition. Array-of parameters are open array parameters. They may look like any array, and they do accept any array, but they get an extra (hidden) parameter, which holds the highest index in the array (the *High* value). Since this is only so in Delphi, and not in C or C++, you'd have a real problem (See also my article on open arrays), since the real number of parameters wouldn't match.

Also, the definition of the array as

```
type
  TPCRL_INFOArray = array of PCRL_INFO;
```

is not the same as the declaration above. It is a dynamic array, which is not the same as a normal array. In fact the dynamic array is closer to the definition of *PPCRL_INFOArray*, since it is also a pointer. But since dynamic arrays are reference counted, and the API doesn't know about reference counts, the reference count might become invalid, and either result in a dynamic array that isn't freed when necessary, or freed to early.

Variable number of parameters

In C and C++, you'll sometimes find functions that can take a variable number of parameters. One example is the *wsprintf* function:

```
int wsprintf(
    LPTSTR lpOut,    /* pointer to buffer for output */
    LPCTSTR lpFmt,   /* pointer to format-control string */
    ...              /* optional arguments */
);
```

The ... (called ellipses) mean, that any number of additional parameters may follow. These are not type checked, and can have any size. The C compiler just pushes them on the stack, and leaves it to the function to interpret the values it pushed. Since the function can't know how many parameters will be passed – at least not when the code of the function is compiled – the caller is responsible for cleaning the stack again, so these function are always *cdecl*.

In Delphi, we have a similar construct, the *array of const* parameter (officially called *variant open array parameter*). The difference is, that the *array of const* is translated to an open *array of TVarRec*, and the values passed are stored according to the type, but not reference counted. Also, like with normal open array parameters, the *High* value is passed (see the aforementioned article). You could create an assembler wrapper function, that uses an array of const, and then translate these and push them on the stack, cleaning the stack after the function call. This is not a trivial matter though. An example can be seen in the JNI translation on the JEDI site.

Fortunately, since Delphi 6, you can declare external functions like *wsprintf* using the *varargs* directive.

```
function wsprintf(lpOut, lpFmt: PChar): Integer; cdecl; varargs;
```

You simply omit the … part, declare the function as *cdecl* and specify *varargs*, as was done above.

Out parameters

If the header was translated with a MIDL compiler, you'll get hints like *[in]* or *[out]* etc. These are in fact quite helpful, since they tell you how the parameter will be used. An *[in, out]* parameter will probably be modified, so you should use *var* for these (please note my proviso above). An *[out]* parameter will be modified too, but can be uninitialized before it is passed. Normally, you could also use *var* here, but not if the parameter is a pointer to an *interface* type (please note, that these are translated as the type itself, since in Delphi these are implicit pointers types, just like objects). If you see an *[out]* or *OUT* comment or macro with an interface type, please declare these as *out* too. This is important for the reference counting mechanism.

So this code (a method of an interface)

```
virtual HRESULT STDMETHODCALLTYPE GetCurMoniker(
    /* [out] */ IMoniker __RPC_FAR *__RPC_FAR *ppimkName) = 0;
```

should be translated as

```
function GetCurMoniker(out ppimkName: IMoniker): HResult;
  stdcall; // for interfaces, virtual is implied
```

If you ignore the for Delphi unimportant *__RPC_FAR* macros above, you see

```
    /* [out] */ IMoniker **ppimkName
```

This can be seen as a pointer to an *IMoniker* pointer. *IMoniker** is the same as *IMoniker* in Delphi, the other * will be translated to *out*, not to *var* here, because of the *[out]* comment, a leftover from the MIDL definition.

Return types

> *One of the main causes of the fall of the Roman Empire was that, lacking zero, they had no way to indicate successful termination of their C programs.* — Robert Firth

A problem I recently encountered was a really hard to find bug. This was caused by something you fortunately hardly ever see in C or C++: returning a structured type (instead of a pointer or reference to one). The code used a struct which looked like:

```
// the list of ABCVar types
enum { VAR_NONE, VAR_FLOAT, VAR_ARRAY, VAR_STRING, VAR_DISP };

// ABCVar is a variant-like structure/union
// that holds any XYZ value
// type member holds variable type (see VAR_ enum above)

typedef struct ABCVar
{
    int type;
    union
    {
        float   val;
        float   *array;
        char    *string;
        void    *disp;
    };
} ABCVar;
```

Note that the size of this structure is 8 bytes: 4 for the *int*, and 4 for the *union*, since all of the types in the union map to 4 bytes, and they overlap.

There were functions *returning* such a type, like:

```
// GetExtraData is optional function for retrieving non-ABC data
```

```
PLUGINAPI ABCVar GetExtraData( LPCTSTR pszTicker,
  LPCTSTR pszName, int nArraySize,
  int nPeriodicity, void* (*pfAlloc)(unsigned int nSize) );
```

I translated this faithfully to the source, and got:

```
type
  {$NODEFINE TpfAlloc}
  TpfAlloc = function(nSize: Cardinal): Pointer cdecl;

// GetExtraData is optional function for retrieving non-ABC data
{$EXTERNALSYM GetExtraData}
function GetExtraData(pszTicker, pszName: PChar;
  nArraySize, nPeriodicity: Integer; pfAlloc: TpfAlloc): ABCVar; cdecl;
```

But soon I received reports of problems with this, and for me, they were hard to track, since I did not have the hardware to test the functions here. It took me a while to see that returning a *struct* like that was uncommon, and that C and Delphi might handle this differently. So I wrote a tiny test program in Visual C++ 2005 Express beta and looked at the resulting assembler code. This showed me that the *ABCVar struct* was returned in the registers *EDX:EAX* (*EDX* with the top 32 bits, and *EAX* with the lower ones). This is not what Delphi does with *record*s at all, not even with *record*s of this size. Delphi treats such return types as extra *var* parameters, and does not return anything (so the *function* is actually a *procedure*).

The only type which Delphi returns as EDX:EAX combination is Int64. So I had to declare an extra type, which I called _ABCVar, as Int64, and tell the users in a large comment why, where and how to cast from _ABCVar to ABCVar and back:

```
 type
   // NOTE: Due to the way functions return structs like ABCVar
   //       in VC++, the type _ABCVar has to be introduced. In
   //       VC++, an ABCVar is returned in register pair EDX:EAX,
   //       while in Delphi, it would be passed by reference. The
   //       only type returned in EDX:EAX in Delphi is Int64. So
   //       the return type in the Delphi translation must be of
   //       the same type, and an _ABCVar must be cast to ABCVar
   //       where necessary.
   //       Fortunately, the problem does not exist with pointers
   //       to ABCVars, or arrays of them.
   //
   //       Function types returning an _ABCVar can't easily be
   //       cast, though:
   //
   //         myABCVar := ABCVar(gSite.AllocArrayResult);
   //
   //       The above will try to cast the function to ABCVar
   //       (which is not valid), and not the function result. To
   //       get the function result, you must explicitly call the
   //       function using parentheses:
   //
   //         myABCVar := ABCVar(gSite.AllocArrayResult());
   //
   //       The above will cast the result of the call to ABCVar,
   //       which is what we want.
   {$NODEFINE _ABCVar}
   _ABCVar = Int64;

   {$NODEFINE PABCVar}
   PABCVar = ^ABCVar;

   {$EXTERNALSYM ABCVar}
    ABCVar = record
     case _type: Integer of
       VAR_FLOAT: (val: Single);
       VAR_ARRAY: (_array: PSingleArray);
       VAR_STRING: (_string: PChar);
       VAR_DISP: (disp: Pointer);
   end;
```

Come to think of it, a better solution would probably have been to define “private” functions that map to the original function in the DLL and are declared to return an Int64, and a function that wraps this call and returns a proper ABCVar.

So, if structs are directly returned from a function, it is wise to have a look at the resulting assembler. Especially since Visual C++ and C++Builder don’t (always) use the same way to return a struct. C++Builder uses a way that is similar to Delphi’s, i.e. it passes the struct as a reference. Additionally, it returns the address of the struct in *EAX*. And, if the *struct* is larger than 8 bytes, Visual C++ will do exactly the same.

Note that the latest Microsoft page about Win64 calling conventions finally describes return values too, for Visual C++.

I am glad to see that Delphi seems to follow that convention too, i.e. records that fit in 64 bits are returned in `RAX`*, larger records are passed as a hidden first* `var` *parameter (in* `RCX`*), and all formal parameters are shifted to the right by one (so the first formal parameter is passed in* `RDX` *or* `XMM1`*), just as the page describes.*

I checked this in Delphi XE2 as well as in Delphi 10.2 Tokyo.

Macro functions

Macro functions are not real functions in C or C++. They are in fact a substitution mechanism, where the arguments to the macro are substituted in the text of the macro. Since these replacements are done by the preprocessor, they are not type safe, they are mere text substitutions. One problem with these macros is, that they may be needed in code, so you'll have to create real functions that do what the macro does. This requires some knowledge of C or C++, since the text can't normally be translated word for word. Also you'll often have to guess the meaning of the parameters, either by looking them up in the documentation, or by looking at the context in which they are used.

You don't have to make these functions *stdcall* or *cdecl*, since they are only functions in your Delphi code. So you can just declare them in the *interface* section of the unit, and implement them in the *implementation* section.

One example:

```
#define GetUrlPolicyPermissions(dw) \
  (dw & URLPOLICY_MASK_PERMISSIONS)
```

(\ is the continuation character for macros) and the conversion in the *implementation* section

```
function GetUrlPolicyPermissions(dw: DWORD): DWORD; inline;
// See note
begin
  Result := dw and URLPOLICY_MASK_PERMISSIONS;
end;
```

Note: since the introduction of the inline directive in Delphi 2005, Delphi uses it for all its macro function translations. It makes sense to do so in your own sources as well. If you have a version before Delphi 2005, you can simply omit the directive. It will not affect the functionality of your code, but perhaps the speed.

But sometimes one can't turn a macro function into a function. In that case the only option is to expand the macro manually, even if that means that you will have to enter a lot of code.

Here are a few nice examples. The first defines a way to construct a constant.

```
#define CTL_CODE(DeviceType,Function,Method,Access) \
  (((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
```

It is used like this:

```
#define IOCTL  CTL_CODE(0x33, 0x0400, METHOD_BUFFERED, FILE_READ_ACCESS)
```

There is no way you can use a function to translate that, so only manual expansion is left:

```
const
  IOCTL = ($33 shl 16) or (FILE_READ_ACCESS shl 14) or ($0400 shl 2) or METHOD_BUFFERED;
```

Another, more convoluted, example defines a range of enumeration items, all with the same prefix:

```
#define CommonAVFunctionEnum( prefix ) \
    prefix ## _Handle,                 \
    prefix ## _VideoSource,            \
    prefix ## _Tune,                  \
                                      \
    /* code snipped */                \
                                      \
    prefix ## _ContextMenu,           \
    prefix ## _DTVResFrameRate,       \
    prefix ## _InputPowerdownTiming,  \
    prefix ## _ProgramInfo,           \
    prefix ## _DiagnosticWindow
```

It is used like this:

```
typedef enum {
    CommonAVFunctionEnum( TVF ),
    TVF_GetWindowHandle = 1000,
    TVF_SendTVWndMessage,

    // code snipped

    TVF_SaveEx,
} TVFunction;
```

The code above defines a large enum type, *TVFunction*, and the first items of that enum are *TVF_Handle*, *TVF_VideoSource*, etc., up to *TVF_DiagnosticWindow*, followed by *TVF_GetWindowHandle*, etc. The only way to translate that is by manually expanding each of the items:

```
type
  TVFunction = (
    TVF_Handle,
    TVF_VideoSource,
    TVF_Tune,

    // Code snipped

    TVF_ContextMenu,
    TVF_DTVResFrameRate,
    TVF_InputPowerdownTiming,
    TVF_ProgramInfo,
    TVF_DiagnosticWindow,
    TVF_GetWindowHandle = 1000,
    TVF_SendTVWndMessage,

    // More code snipped

    TVF_SaveEx);
```

Records and alignment

Records are are an important part of an API, since they define the OS structures the API functions operate on. Because of this, the record conversion must result in an exact binary copy of the original structure. Most problems with structure conversions happen because of bad alignment.

Alignment is done to make access to some variables, or members of a struct (The Delphi term is: fields of a record) faster for the processor. A 32 bit (4 byte) type can be acccessed faster by the processor if it is aligned on a 4 byte boundary, i.e. if the address is divisible by 4 without a rest. An 8 byte type should ideally be aligned on an 8 byte boundary, etc. Note that I am talking about simple types here, not about compound types like arrays or records.

An aligned *struct* will contain some extra filler bytes to make sure that the members are aligned on “their” boundaries. This way a struct can be larger than the sum of the sizes of the composing types. If alignment is set to a smaller value, only up to that many filler bytes will be inserted, so if alignment is set to 4, an 8 byte type will be aligned on the nearest 4 byte boundary, which is not necessarily an 8 byte boundary. If alignment is set to 1, no filling will occur at all, and the sum of the member sizes is the same as the size of the *struct*. This is equivalent to a *packed record* in Delphi.

Note: filler bytes are also inserted after the last field of a record, to align the next record, for instance in an array, as well.

The default packing for Win32 headers is, as far as I can tell, 8 byte. So types that should be aligned on 8 byte boundaries are aligned on 8 bytes, but types that only need a 4 byte alignment are aligned on such a boundary, etc. Unfortunately not all structs in headers are aligned like that. Most C and C++ compilers have the capability to set the alignment to 1, 2, 4, 8 or 16 bytes. But how they do this is not always uniform.

#pragma pack

The most popular way to set the alignment is to use a line like:

```
#pragma pack(push, 4)
```

That sets the alignment to 4 byte boundaries, so all types of 4 byte or larger will be aligned on 4 bytes, smaller types will be aligned on 2 or 1 byte, according to their types. The *push* means that the current alignment setting is saved somewhere before setting the 4 byte alignment. Later on, the previous alignment can be restored with

```
#pragma pack(pop)
```

So to find out what the alignment for a *struct* is, you should try to find the last occurrences of `#pragma pack`. Note that this can be in an included header file as well.

Other ways

Other ways of setting or resetting the packing is

- by external compiler options like *-a8* e.g. in a make file;
- in other pragmas, like `#pragma option push -a8` (8 byte) or `#pragma option push -a-` (1 byte);
- by including special headers, like *pshpack8.h* — `pack(push, 8)` and *poppack.h* — `pack(pop)`.

I will not discuss all the options and headers. Some of these may vary with the compiler, although most are probably compatible with Microsoft's Visual C++. More info can be found in the Win32 (and Win64) SDK help files and online in the MSDN library.

Using a C++ compiler

If you have access to a C++ compiler, it is often useful to let that compiler tell you the size of a struct. This is usually a good indication of the alignment setting. If you don't have one, you can use one of the various free C++ compilers, for instance a GNU compiler, the free C++Builder command line compiler 5.5 from Embarcadero, or the free Visual C++ Express compiler from Microsoft (note that this requires an installation of .NET).

These days, you can also use the free C++Builder 10.2 Tokyo Community Edition. It is equal to C++Builder 10.2 Tokyo Professional, it just has a different license.

This simple code demonstrates how you can test the alignment of a struct. The comments give the offsets and sizes of the members for that alignment size.

```
/* structsize.c */
#include <stdio.h>
#include <conio.h>

#pragma pack(1)
struct X1
{                         /* ofs + size */
    unsigned char B;      /* 0   + 1 */
    unsigned short S1;    /* 1   + 2 */
    unsigned short S2;    /* 3   + 2 */
    unsigned long L;      /* 5   + 4 */
    double D;             /* 9   + 8 */
    long L2;              /* 17  + 4 */
    long double E;        /* 21  + 10 */
};                        /* 31 */

#pragma pack(2)
struct X2
{                         /* ofs + size */
    unsigned char B;      /* 0   + 1 */
    unsigned short S1;    /* 2   + 2 */
    unsigned short S2;    /* 4   + 2 */
    unsigned long L;      /* 6   + 4 */
    double D;             /* 10  + 8 */
    long L2;              /* 18  + 4 */
    long double E;        /* 22  + 10 */
};                        /* 32 */

```

```
#pragma pack(4)
struct X4
{                           /* ofs + size */
    unsigned char B;        /* 0   + 1 */
    unsigned short S1;      /* 2   + 2 */
    unsigned short S2;      /* 4   + 2 */
    unsigned long L;        /* 8   + 4 */
    double D;               /* 12  + 8 */
    long L2;                /* 20  + 4 */
    long double E;          /* 24  + 10 */
};                          /* 36 */

#pragma pack(8)
struct X8
{                           /* ofs + size */
    unsigned char B;        /* 0   + 1 */
    unsigned short S1;      /* 2   + 2 */
    unsigned short S2;      /* 4   + 2 */
    unsigned long L;        /* 8   + 4 */
    double D;               /* 16  + 8 */
    long L2;                /* 24  + 4 */
    long double E;          /* 32  + 10 */
};                          /* 48 */

int main()
{
    printf("%d\n", sizeof(struct X1));
    printf("%d\n", sizeof(struct X2));
    printf("%d\n", sizeof(struct X4));
    printf("%d\n", sizeof(struct X8));
    getch();

    return 0;
}
```

This will produce the expected output (see the comments in the code above):

```
31
32
36
48
```

Getting alignment right

The problem is creating records with exactly the same alignment in Delphi. In Delphi version 6 or higher, it is quite simple. These versions have an alignment system that is compatible with Microsoft Visual C++. You can specify the alignment in a similar way as in C and C++ compilers, using directives like `{$ALIGN 4}` or `{$A8}`. The online help for Delphi describes them in detail.

Checking sizes and alignments

If you want to know exactly how the members are aligned, you can use C++, the *offsetof()* macro in *stddef.h* and the *typeid()* RTTI function.

The C++ part

To make things a little easier, I wrote a function and a few macros. I put them in a header file, which I include in my test program. Here is the header file, *members.h*:

```
#ifndef MEMBERS_H
#define MEMBERS_H

void print_member(const char *type, const char *name, int offset)
{
    char buffer[256];
    sprintf(buffer, "    %s %s;", type, name);
    printf("%-48s // offset %d\n", buffer, offset);
}

#define print(member) \
    print_member(typeid(((type *)0)->member).name(), #member, offsetof(type, member))

#define start \
    printf("struct %-42s/\/ size %d\n{\n", typeid(type).name(), sizeof(type))

#define end \
    printf("};\n\n");

#endif
```

Now I am aware of the fact that this will make C++ purists cringe, and have been looking for a more elegant solution, but no one I asked was able to come up with one, yet. They talked about using templates and type traits, etc. but could not provide a solution. So now I use this to display the offsets. To do this, I must first `#define type` as the struct type I want to investigate. See below.

```
#include <stdio.h>      /* includes stddef.h */
#include "windows.h"    /* for the struct type I want to inspect */
#include "members.h"

int main()
{
    #define type SECURITY_DESCRIPTOR
    start;
        print(Revision);
        print(Sbz1);
        print(Control);
        print(Owner);
        print(Group);
        print(Sacl);
        print(Dacl);
    end;
    #undef type
}
```

Note that you must `#undef type` so you can re-define it for any new structs. You do this for each struct you want to examine.

Of course, in your code, you would include the header, and specify the members of the *struct* you want to translate. The output of this little example is:

```
struct _SECURITY_DESCRIPTOR                     // size 20
{
    unsigned char Revision;                     // offset 0
    unsigned char Sbz1;                         // offset 1
    unsigned short Control;                     // offset 2
    void * Owner;                               // offset 4
    void * Group;                               // offset 8
    _ACL * Sacl;                                // offset 12
    _ACL * Dacl;                                // offset 16
};
```

You have to invoke *print()* for each member you want to inspect. I don't know of a way to make C++ display the entire struct with a single command. Perhaps I can use the Boost Reflect library.

You can only do this with C++. C does not have the RTTI function `typeid()`*, which is needed to get the type names of the members, e.g. with* `typeid(((structname*)0)->membername).name()`.

The Delphi counterpart

To check if you have the offsets in your translation right, you can use RTTI, if your version has the unit *System.Rtti*. You can write a little function to display the offsets of your translated records and check if they are the same as in the output of the C++ program:

```
procedure WriteOffsets(Info: Pointer);
var
  LContext: TRttiContext;
  LType: TRttiType;
  LField: TRttiField;
begin
  LType := LContext.GetType(Info);
  if Assigned(LType) then
    if LType.TypeKind = tkRecord then
    begin
      Writeln('type');
      Writeln(('  ' + LType.QualifiedName + ' = record ').PadRight(48),
        ' // Size = ', LType.TypeSize);
      for LField in LType.GetFields do
```

```
        Writeln(('    ' + LField.Name + ': ' + LField.FieldType.Name + ';').PadRight(48),
          ' // ', LField.Offset);
      Writeln('  end;');
      Writeln;
    end
    else
    begin
      Writeln(LType.QualifiedName, ' is not a record');
    end
  else
    Writeln('Invalid type');
end;
```

You use it like this:

```
program DemoOffsets;

{$APPTYPE CONSOLE}

uses
  Velthuis.Offsets,                             // For WriteOffsets
  Winapi.Windows;                               // Contains the type I want to inspect:
                                                //   TSecurityDescriptor
begin
  WriteOffsets(TypeInfo(TSecurityDescriptor));
  // WriteOffsets(TypeInfo(WhateverStruct));
end.
```

and the output is (in Win32):

```
type
  Winapi.Windows._SECURITY_DESCRIPTOR = record   // size 20
    Revision: Byte;                              // offset 0
    Sbz1: Byte;                                  // offset 1
    Control: Word;                               // offset 2
    Owner: Pointer;                              // offset 4
    Group: Pointer;                              // offset 8
    Sacl: PACL;                                  // offset 12
    Dacl: PACL;                                  // offset 16
  end;
```

Now you can use such outputs to compare the offsets of your conversion to the output of the C++ program.

Note: To check your records, you can usually get along by comparing the sizes given by your C and Delphi test programs. Only if these differ, you should check the alignments too.

Adding filler bytes

In a version that does not allow specifying the exact alignment, or in versions that don't always align properly, you can add filler bytes of your own, if necessary. To make sure that Delphi doesn't add its own alignment fill bytes, you can use *packed record*. Packed records are always unaligned, so you can do your own manual aligning. The *X8* struct above, which I designed such that it has a number of gaps, would translate to:

```
type
  PX8 = ^X8;
  tagX8 = packed record          // offset of next element
    B: Byte;                     // 1
    Fill1: Byte;                 // 2
    S1: Word;                    // 4
    S2: Word;                    // 6
    Fill2: array[7..8] of Byte;  // 8
    L: Longword;                 // 12
    Fill3: array[13..16] of Byte; // 16
    D: Double;                   // 24
    L2: Longint;                 // 28
    Fill4: array[29..32] of Byte; // 32
    E: Extended;                 // 42
    Fill5: array[43..48] of Byte; // 48
  end;
  X8 = tagX8;
```

In Delphi 6 or higher, you should not use a *packed record*, but rather use the alignment declared in the header and a normal *record*. E.g. in *ShellAPI.h*, the declared alignment is 1 for Win32, but 8 for Win64. So to translate *ShellAPI.h*, you should put somewhere near the top of your conversion unit:

```
{$IFDEF WIN32}
  {$ALIGN 1}
{$ELSE} // Win64
  {$ALIGN 8}
{$ENDIF}
```

The packed record with manual filler bytes would of course also work in Delphi 6 or higher, but has a disadvantage: if you use a packed record in another, aligned record, the packed record will always be byte-aligned.

One nice example of this is the TRect structure in Delphi 6 and Delphi 7. In these and some later versions, it is declared as packed record, although it shouldn't be. For normal use, this is not really a problem, since it contains 4 Integers, so it is 16 bytes in size anyway. It contains no filler bytes. But if you use a TRect like that in another record, you can get unexpected problems. For instance this record (still assuming that TRect is packed, which is not true in later versions of Delphi anymore):

```
{$ALIGN 8}
type
  TMyRec = record
    B: Byte;
    R: TRect;
  end;
```

will be 17 bytes in size, and not, as it ought to be, 20 bytes. The TRect is not aligned on a 4 byte boundary, although it contains four Integers. If it hadn't been declared as packed record, it would have been aligned properly. A test with a simple C program confirms that the size should have been 20, and not 17.

So if you can, use the declared alignment, and plain, unpacked *record*. If your alignments or sizes differ from the aligment found in your C program, check if you didn't forget a field, if the fields are all of the right type and if you used the correct alignment (I found out that I hadn't done that for *ShellAPI.h* because the alignments and size didn't match). Only if you checked all these things and there is still a size or alignment mismatch, you should think of manually adding filler bytes.

Unions

In C, unions are similar to structs, but each of the "fields" occupies the same place in memory. Delphi has a similar construct, variant records (do not confuse these with the *Variant* type), the records with the `case .. of` parts. But there are a few differences. In a union, each of the fields overlaps with each of the other. So there can't be two fields together, or they must be declared as a struct. In Delphi, several fields can be grouped together under one of the case selectors with parentheses

Note: as far as I know, the C standard does not guarantee that all fields of a union are aligned on the same address, but usually, especially on Windows, you can depend on it.

But there is one bigger problem: C unions can be anonymous. This is a problem since in Delphi, you can't have anonymous fields. Here is an example from *wincrypt.h* (I removed the comments):

```
typedef struct _CMC_STATUS_INFO {
    DWORD       dwStatus;
    DWORD       cBodyList;
    DWORD       *rgdwBodyList;
    LPWSTR      pwszStatusString;   // OPTIONAL
    DWORD       dwOtherInfoChoice;
    union  {
        // CMC_OTHER_INFO_NO_CHOICE
        //  none
        // CMC_OTHER_INFO_FAIL_CHOICE
        DWORD                   dwFailInfo;
        // CMC_OTHER_INFO_PEND_CHOICE
        PCMC_PEND_INFO          pPendInfo;
    };
} CMC_STATUS_INFO, *PCMC_STATUS_INFO;
```

The union is at the end of the struct, and apparently the *dwOtherInfoChoice* field is a selector, so you can translate it like:

```
type
  _CMC_STATUS_INFO = record
    dwStatus: DWORD;
    cBodyList: DWORD;
    rgdwBodyList: ^DWORD;
    pwszStatusString: LPWSTR;   // OPTIONAL
    case dwOtherInfoChoice: DWORD of
      CMC_OTHER_INFO_NO_CHOICE: ({none});
      CMC_OTHER_INFO_FAIL_CHOICE: (dwFailInfo: DWORD);
      CMC_OTHER_INFO_PEND_CHOICE: (pPendInfo: PCMCPendInfo);
  end;
```

But it gets tricky if the union is somewhere in the middle of the struct, as in this example, also from *wincrypt.h*:

```
typedef struct _OCSP_BASIC_RESPONSE_INFO {
    DWORD                       dwVersion;
    DWORD                       dwResponderIdChoice;
    union {
        // OCSP_BASIC_BY_NAME_RESPONDER_ID
        CERT_NAME_BLOB              ByNameResponderId;
        // OCSP_BASIC_BY_KEY_RESPONDER_ID
        CRYPT_HASH_BLOB              ByKeyResponderId;
    };
    FILETIME                    ProducedAt;
    DWORD                       cResponseEntry;
    POCSP_BASIC_RESPONSE_ENTRY  rgResponseEntry;
    DWORD                       cExtension;
    PCERT_EXTENSION             rgExtension;
} OCSP_BASIC_RESPONSE_INFO, *POCSP_BASIC_RESPONSE_INFO;
```

The problem is that Delphi variant records must always have the case part at the end, since the `end` will end the `case` part as well as the record declaration. But there are a few fields, like *ProducedAt* and *cResponseEntry*, that come *after* the union. So how do I translate this?

One solution would be to turn the union into an embedded record. But that would mean that this embedded record would get its own field name:

```
type
  _OCSP_BASIC_RESPONSE_INFO = record
    dwVersion: DWORD;
    dwResponderIdChoice: DWORD;
    union1: record
      case DWORD of
        OCSP_BASIC_BY_NAME_RESPONDER_ID:
          (ByNameResponderId: CERT_NAME_BLOB); // size: 8
        OCSP_BASIC_BY_KEY_RESPONDER_ID:
          (ByKeyResponderId: CRYPT_HASH_BLOB;  // size: 8
    end;
    ProducedAt: FILETIME;
    cResponseEntry: DWORD;
    rgResponseEntry: POCSP_BASIC_RESPONSE_ENTRY;
    cExtension: DWORD;
    rgExtension: PCERT_EXTENSION
  end;
```

The disadvantage is that you can't access the *ByResponderNameId* and *ByKeyResponderId* items directly anymore. You must prefix them with *union1*, i.e. now you must use *MyRespInfo.union1.ByResponderNameId* instead of *MyRespInfo.ByResponderNameId*.

I found a rather tricky solution which does not require this. Consider that the union occupies the same size as the largest item in it. Following fields in the struct will be aligned after the largest item in the union. So you can just as well add all following fields to the largest variant part of your translated record, and they will be at exactly the same offset in the record as they would be with the *union1* solution above:

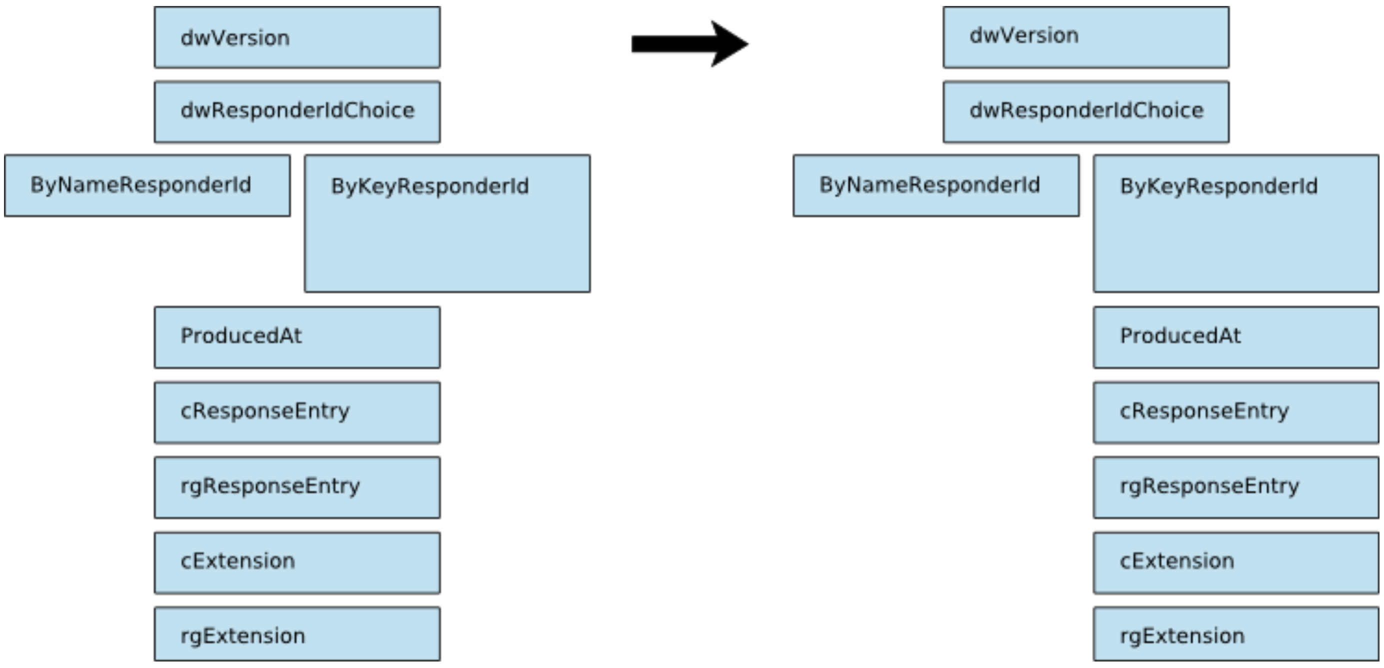
```
type
  _OCSP_BASIC_RESPONSE_INFO = record
    dwVersion: DWORD;
    case dwResponderIdChoice: DWORD of
    // union {
      OCSP_BASIC_BY_NAME_RESPONDER_ID:
        (ByNameResponderId: CERT_NAME_BLOB); // size: 8
      OCSP_BASIC_BY_KEY_RESPONDER_ID:
        (ByKeyResponderId: CRYPT_HASH_BLOB;  // size: 8
    // }
    ProducedAt: FILETIME;
    cResponseEntry: DWORD;
    rgResponseEntry: POCSP_BASIC_RESPONSE_ENTRY;
    cExtension: DWORD;
    rgExtension: PCERT_EXTENSION
    );
    end;
```

In other words, the last variant part is now (reformatted a bit):

```
        (ByKeyResponderId: CRYPT_HASH_BLOB;
    // }
    ProducedAt: FILETIME;
    cResponseEntry: DWORD;
    rgResponseEntry: POCSP_BASIC_RESPONSE_ENTRY;
    cExtension: DWORD;
    rgExtension: PCERT_EXTENSION
    );
```

Now you can access the union parts directly again, for instance as *MyRespInfo.ByResponderNameId*.

In the following diagram, I try to show the differences. On the left, the union parts *ByNameResponderId* and *ByKeyResponderId* have the same offset in memory, but one is larger than the other. As you can see, the following fields of the struct (*ProducedAt* etc.) come after the largest of the union items. In the right version, I show how the fields after the union have been arranged under the second, the largest union item, *ByKeyResponderId*. As you can see, they keep the same offset, and that is all that counts.



(I cheated. In reality both items in the union are the same size: 8 byte. Making them different sizes much better demonstrates the principle, though.)

This does require that you find the largest item in the union, and use that as the item in the case part to which you add all following fields, even if they are originally not part of the union. But to be sure, you should always test the assumptions you make, with a little C program giving the offsets of the items after the union and a little Delphi program doing the same.

To find the largest item of a union, I write a small C program that gives me the sizes of the types in the union, if I don't know them (if all have the same type, or all are pointers, that is strictly not necessary).

Note that this trick will also work if there are multiple unions:

```
typedef struct _CERT_SYSTEM_STORE_RELOCATE_PARA {
    union {
        HKEY                hKeyBase;
        void                *pvBase;
    };
    union {
        void                *pvSystemStore;
        LPCSTR              pszSystemStore;
        LPCWSTR             pwszSystemStore;
    };
};
```

This will become (omitted pointer type and externalsym):

```
_CERT_SYSTEM_STORE_RELOCATE_PARA = record
  case Byte of
    0: (hKeyBase: HKEY);
    1: (pvBase: Pointer;
         case Byte of
           0: (pvSystemStore: Pointer);
           1: (pszSystemStore: PAnsiChar);
           2: (pwszSystemStore: PWideChar);
       );
end;
```

UPDATE! *Apparently this was not an original idea of mine. In CommCtrl.pas you'll find a nice solution which also handles multiple unions in one record and does not require the use of embedded (named) records.*

It turns out that you can have another case selector inside one of the branches of a variant. This sounds a bit confusing, but just look at the (expanded) declaration of _PROPSHEETPAGEA_V2 in prsht.h (for property sheets, which is included in commctrl.h), which follows:

```
typedef struct _PROPSHEETPAGEA_V2
{
    DWORD           dwSize;
    DWORD           dwFlags;
    HINSTANCE       hInstance;
    union
    {
        LPCSTR      pszTemplate;
        PROPSHEETPAGE_RESOURCE pResource;
    } DUMMYUNIONNAME;
    union
    {
        HICON        hIcon;
        LPCSTR       pszIcon;
    } DUMMYUNIONNAME2;
    LPCSTR           pszTitle;
    DLGPROC          pfnDlgProc;
    LPARAM           lParam;
    LPFNPSPCALLBACKA pfnCallback;
    UINT             *pcRefParent;
    LPCSTR           pszHeaderTitle;    // this is displayed in the header
    LPCSTR           pszHeaderSubTitle; //
} PROPSHEETPAGEA_V1, *LPPROPSHEETPAGEA_V1;
```

DUMMYUNIONNAME and DUMMYUNIONNAME2 are macros that amount to nothing, so the two unions are anonymous. CodeGear's translation uses the same idea as described before, but does it for both unions:

```
type
  _PROPSHEETPAGEA = record
    dwSize: Longint;
    dwFlags: Longint;
    hInstance: THandle;
    case Integer of
      0: (
        pszTemplate: PAnsiChar);
      1: (
        pResource: Pointer;
        case Integer of
          0: (
            hIcon: THandle);
          1: (
            pszIcon: PAnsiChar;
            pszTitle: PAnsiChar;
            pfnDlgProc: Pointer;
            lParam: Longint;
            pfnCallback: TFNPSPCallbackA;
            pcRefParent: PInteger;
            pszHeaderTitle: PAnsiChar;      // this is displayed in the header
            pszHeaderSubTitle: PAnsiChar)); //
  end;
```

As you see, there is a second case selector in the last variant part of the first, i.e. as if that variant part were a record of its own. This way it is easy to cascade such union declarations, and the problem in the Info box above is solved.

Bitfields

Sometimes you’ll encounter a declaration like the following:

```
typedef struct _RTCP_RECEIVER_REPORT {
    DWORD FractionLost:8;
    DWORD TotalLostPackets:24;
    DWORD HighestSequenceNum;
    DWORD InterarrivalJitter;
    DWORD LastSRTimestamp;
    DWORD DelaySinceLastSR;
} RTCP_RECEIVER_REPORT, *PRTCP_RECEIVER_REPORT;
```

The problem with the above declaration is, that it contains C bitfields. *FractionLost* and *TotalLostPackets* are not *DWORD*s at all. The `:8` following the identifier *FractionLost* means that FractionLost only occupies 8 bits in a DWORD, and `:24` means that *TotalLostPackets* contains the remainging 24 bits. So *together* they occupy one *DWORD*, not each one of its own. Such items are very hard to translate. *Note: translation is platform dependent as well, but since this article is primarily for Win32, or at least Intel, you can disregard that.*

I would translate this as:

```
type
  PRTCP_RECEIVER_REPORT = ^RTCP_RECEIVER_REPORT;
  _RTCP_RECEIVER_REPORT = record
    FractionLost_TotalLostPackets: DWORD;
      // DWORD FractionLost:8
      // DWORD TotalLostPackets:24
    HighestSequenceNum: DWORD;
    InterarrivalJitter: DWORD;
    LastSRTimestamp: DWORD;
    DelaySinceLastSR: DWORD;
  end;
  RTCP_RECEIVER_REPORT = _RTCP_RECEIVER_REPORT;
```

FractionLost could have been translated as a Byte, but that would not solve the problem with *TotalLostPackets*, since there are no 3 byte types that could be used for it. This way, if you attempt to access either member, you will get a message that the identifier is not defined, and have to look for yourself. Often, you will have many items of 1 or 2 bits in size, so you have the chance to be creative with your identifiers.

There is no way you can declare the items in Delphi thus, that they are readily accessible. You could provide constants for the shift and the mask to access the single parts of the *DWORD*, or even an enumerated type and a function to access the parts, but that is up to you. The user will most of the time have to do something extra to access the bitfields.

An example for constants to acess the bitfields:

```
// Assumes first "shr", then "and"
const
  // shifts for FractionLost_TotalLostPackets field
  shrFractionLost = 0;
  shrTotalLostPackets = 8;

  // masks for FractionLost_TotalLostPackets field
  andFractionLost = $000000FF;
  andTotalLostPackets = $00FFFFFF;
```

Now the *TotalLostPackets* bitfield can be accessed like:

```
LostPackets := Report.FractionLost_TotalLostPackets
                 shr shrTotalLostPackets
                 and andTotalLostPackets;
```

Of course, you can also write access functions for each of the bitfields, something like:

```
function FractionLost(const Report: RTCP_RECEIVER_REPORT): Byte; overload;
begin
  Result := Report.FractionLost_TotalLostPackets and $000000FF;
end;

procedure FractionLost(var Report: RTCP_RECEIVER_REPORT; Value: Byte); overload;
begin
  Report.FractionLost_TotalLostPackets :=
    (Report.FractionLost_TotalLostPackets and $FFFFFF00) or Value;
end;

// etc...
```

It is used like:

```
var
  Fraction: Integer;
  R: RTCP_RECEIVER_REPORT;
begin
  Fraction := FractionLost(R);
  FractionLost(R, Fraction * 2);
```

Records with properties

In Delphi 2006 for Win32 (as used in BDS2006 and Turbo Delphi 2006), records with methods, operators and properties were introduced. This allows you to define the bitfields as properties and define getter and setter methods:

```
type
  PRTCP_RECEIVER_REPORT = ^RTCP_RECEIVER_REPORT;
  _RTCP_RECEIVER_REPORT = record
  private
    FractionLost_TotalLostPackets: DWORD;
      // DWORD FractionLost:8
      // DWORD TotalLostPackets:24
    function GetFractionLost: Byte;
    procedure SetFractionLost(Value: Byte);
    function GetTotalLostPackets: DWORD;
    procedure SetTotalLostPackets(Value: DWORD);
  public
    HighestSequenceNum: DWORD;
    InterarrivalJitter: DWORD;
    LastSRTimestamp: DWORD;
    DelaySinceLastSR: DWORD;

    property FractionLost: Byte read GetFractionLost
                                write SetFractionLost;
    property TotalLostPackets: DWORD read GetTotalLostPackets
                                     write SetTotalLostPackets;
  end;
  RTCP_RECEIVER_REPORT = _RTCP_RECEIVER_REPORT;
```

In the implementation section, you can define the methods to get and set the properties:

```
{ _RTCP_RECEIVER_REPORT }

function _RTCP_RECEIVER_REPORT.GetFractionLost: Byte;
begin
  Result := FractionLost_TotalLostPackets and $FF;
end;

function _RTCP_RECEIVER_REPORT.GetTotalLostPackets: DWORD;
begin
  Result := (FractionLost_TotalLostPackets shr 8) and $00FFFFFF;
end;

procedure _RTCP_RECEIVER_REPORT.SetFractionLost(Value: Byte);
begin
  FractionLost_TotalLostPackets :=
    FractionLost_TotalLostPackets and $FFFFFF00 or Value;
end;

procedure _RTCP_RECEIVER_REPORT.SetTotalLostPackets(
  Value: DWORD);
begin
  FractionLost_TotalLostPackets :=
    (FractionLost_TotalLostPackets and $000000FF) or
    (Value and $00FFFFFF) shl 8;
end;
```

A short test shows that this works as expected. If there are many bitfields, it may be a lot of work, though, and it will not work in versions of Delphi below 2006.

Using the property index

On stackoverflow.com, Patrick van Logchem has an excellent and elegant proposal. He does not write a property getter and setter for each bitfield, but instead, he uses the indexing of properties. The index of a property is not necessarily a real index into a structure; if you use getters and setters, it is simply a parameter to those functions. Now, he uses the lower two bytes of the index to indicate offset and number of bits respectively (well, actually not in that article, but in a post to the JEDI API conversion newsgroup at news://forums.talkto.net/jedi.apiconversion). He then only has to define two general routines that get or set bits using such an index value, and use that in only one getter and setter routine for each record.

Here is a slightly reformatted version of his code for the routines:

```
function GetDWordBits(const Bits: DWORD; const aIndex: Integer): Integer;
begin
  Result := (Bits shr (aIndex shr 8))       // offset
            and ((1 shl Byte(aIndex)) - 1); // mask
end;

procedure SetDWordBits(var Bits: DWORD; const aIndex: Integer; const aValue: Integer);
var
  Offset: Byte;
  Mask: Integer;
begin
  Mask := ((1 shl Byte(aIndex)) - 1);
  Assert(aValue <= Mask);

  Offset := aIndex shr 8;
  Bits := (Bits and (not (Mask shl Offset)))
          or DWORD(aValue shl Offset);
end;
```

So what does `((1 shl Byte(aIndex)) - 1)` *exactly mean? If you need, say, 8 bits, the mask should be $FF. Well, 1 shl 8 is $100, and minus 1, that becomes $FF.*

So now, instead of a getter and setter for each bitfield property, he only uses one of each, like this:

```
type
  RLDT_ENTRY_Bits = packed record
  private
    Flags: DWord;
    function GetBits(const aIndex: Integer): Integer;
    procedure SetBits(const aIndex: Integer; const aValue: Integer);
  public
    property BaseMid: Integer index $0008 read GetBits write SetBits;     // 8 bits at offset 0
    property _Type: Integer index $0805 read GetBits write SetBits;       // 5 bits at offset 8 ($08)
    property Dpl: Integer index $0D02 read GetBits write SetBits;         // 2 bits at offset 13 ($0D)
    property Pres: Integer index $0F01 read GetBits write SetBits;        // 1 bit  at offset 15 ($0F)
    property LimitHi: Integer index $1004 read GetBits write SetBits;     // 4 bits at offset 16 ($10)
    property Sys: Integer index $1401 read GetBits write SetBits;         // 1 bit  at offset 20 ($14)
    property Reserved_0: Integer index $1501 read GetBits write SetBits;  // 1 bit  at offset 21 ($15)
    property Default_Big: Integer index $1601 read GetBits write SetBits; // 1 bit  at offset 22 ($16)
    property Granularity: Integer index $1701 read GetBits write SetBits; // 1 bit  at offset 23 ($17)
    property BaseHi: Integer index $1808 read GetBits write SetBits;      // 8 bits at offset 24 ($18)
  end;
```

As you can see, the properties all use the same getter and setter functions, *GetBits* and *SetBits* respectively, and only distinguish themselves by the index used for them. The comments demonstrate how the index is coded.

The getter and setter simply look like:

```
function RLDT_ENTRY_Bits.GetBits(const aIndex: Integer): Integer;
begin
  Result := GetDWordBits(Flags, aIndex);
end;

procedure RLDT_ENTRY_Bits.SetBits(const aIndex: Integer; const aValue: Integer);
begin
  SetDWordBits(Flags, aIndex, aValue);
end;
```

Enums

C and C++ *enum*s are similar to Delphi enumerated types. They denote a range of named identifiers, each with a different value. In C and C++, you can set the values of each identifier, so you can have enums like (in *mshtml.h*):

```
typedef
enum _htmlDesignMode
{   htmlDesignModeInherit = -2,
    htmlDesignModeOn      = -1,
    htmlDesignModeOff     = 0,
    htmlDesignMode_Max    = 2147483647L
} htmlDesignMode;
```

The values don't start at 0, and there is a huge gap. In Delphi 6 or higher, you can declare an enumeration type like that as well:

```
type
  _htmlDesignMode = (
    htmlDesignModeInherit = -2,
    htmlDesignModeOn      = -1,
    htmlDesignModeOff     = 0,
    htmlDesignMode_Max    = 2147483647
  );
  htmlDesignMode = _htmlDesignMode;
```

***Update:** While it may be convenient to translate such enums as Delphi enumerated types, I recently stumbled over a difference. In C and C++, enumerated types are often assignment-compatible with normal integral types, like int or unsigned int. Many function declarations reflect this, and simply declare parameters expecting such enumerated values as DWORD, ULONG, or similar, and not of the special enum type expected. In C, this generally causes no problems. In Delphi, enumerated types are a type of their own, and can not be assigned to normal integral types or to other enumerated types, so you would have to cast such enumerated values to DWORD. You could of course also declare the parameters as the correct enum type, but this may cause incompatibilities with existing code, especially if this relies on an older translation.*

So in many cases, it is probably best if you don't use Delphi's enumerated types at all for such conversions, but declare the enum as DWORD or Longint, and the values as simple untyped constants, as described below, for Delphi versions lower than 7.

In versions before Delphi 6, you'll have to use constants instead, and define a type for the enum. You get something like:

```
type
  _htmlDesignMode = Longint;
  htmlDesignMode = _htmlDesignMode;

const
  htmlDesignModeInherit = -2;
  htmlDesignModeOn      = -1;
  htmlDesignModeOff     = 0;
  htmlDesignMode_Max    = 2147483647;
```

Bit values

In C header files, enums are sometimes abused to define values for flags or bits in a value used as bitset. An example from shlobj.h:

```
typedef enum _tagAUTOCOMPLETELISTOPTIONS
{
    ACLO_NONE         = 0,  // don't enumerate anything
    ACLO_CURRENTDIR   = 1,  // enumerate current directory
    ACLO_MYCOMPUTER   = 2,  // enumerate MyComputer
    ACLO_DESKTOP      = 4,  // enumerate Desktop Folder
    ACLO_FAVORITES    = 8,  // enumerate Favorites Folder
    ACLO_FILESYSONLY  = 16, // enumerate only the
                            // file system
    ACLO_FILESYSDIRS  = 32, // enumerate only the
                            // file system dirs, UNC
                            // shares, and UNC servers.
} AUTOCOMPLETELISTOPTIONS;

// Note: the comma after the last value, 32, is valid C and C++.
// It is present in the original header file.
```

These should **not** be translated as enumerated type, not even in Delphi 6 or higher, because most of the time they are used to select bits using the C equivalent of the bitwise *or* or *and* operators. This is possible with enums in C, but not in Delphi. That is why you should *always* convert such values, which typically — but not always — represent powers of 2, as constants.

```
 type
   _tagAUTOCOMPLETELISTOPTIONS = Byte;
   AUTOCOMPLETELISTOPTIONS = _tagAUTOCOMPLETELISTOPTIONS;

 const
   ACLO_NONE        = 0;  // don't enumerate anything
   ACLO_CURRENTDIR  = 1;  // enumerate current directory
   ACLO_MYCOMPUTER  = 2;  // enumerate MyComputer
   ACLO_DESKTOP     = 4;  // enumerate Desktop Folder
   ACLO_FAVORITES   = 8;  // enumerate Favorites Folder
   ACLO_FILESYSONLY = 16; // enumerate only the file system
   ACLO_FILESYSDIRS = 32; // enumerate only the file system
                          // dirs, UNC shares, and UNC servers
```

Type

Don't be tempted to make such values typed constants, like

```
const
  // don't enumerate anything
  ACLO_NONE: AUTOCOMPLETELISTOPTIONS = 0;
```

In Delphi, typed constants do have a type and a size (they are stored like global variables), but they have one disadvantage: you can't use them in constant expressions. This means you can't define other constants as combinations of them, or use them as label for a case statement. Always use them as simple constant.

If you really want to give such values a type, you can do it with a trick:

```
const
  INVALID_HANDLE_VALUE = DWORD(-1);
```

This will give *INVALID_HANDLE_VALUE* the associated type *DWORD*. This means that if you do:

```
procedure OpenAFile(Handle: Longint);
begin
  if Handle <> INVALID_HANDLE_VALUE then

    // etc...
```

You will get a warning that the expression always evaluates to *False*, since *INVALID_HANDLE_VALUE* is a *DWORD* with value *$FFFFFFFF*, which is always more than any *Longint* value could be.

Size

According to the C++ standard, the size of an enum is dependent on the compiler. This means that one compiler can choose to use the smallest type necessary to be able to hold all values in the enum, while another compiler can make all integer-sized (or even larger, if the contained values require it).

It turns out that Microsoft's compilers use integer sized enums, while Borland/CodeGear/Embarcadero's use the smallest possible type required. Because most DLL providers will try to be compatible with Microsoft, you can probably safely assume that enums are 4 bytes in size. Most headers Embarcadero provides with its C++ compilers are modified to set that option for the Borland compiler as well.

By default, Delphi, like the C++ compilers mentioned, uses the smallest necessary type to accomodate all values, so to be compatible with Microsoft, you will have to set a compiler directive, which makes all Delphi enums 4 bytes in size as well:

```
{$MINENUMSIZE 4}
```

If you have an older version of Delphi, you can use the following instead:

```
{$Z+}
```

If your version of Delphi doesn't have either option, your best choice is to declare the enum values as constants, and to declare the type as DWORD.

Translating types

There are different kinds of (integer) types in Delphi. The Delphi help distinguishes between generic types and fundamental types. The generic types are portable, but their size is platform dependent. They are *Integer* and *Cardinal*, which are both 16 bit in Delphi 1 (for 16 bit Windows) and 32 bit in 32 bit versions of Delphi (actually, Cardinal used to be officially 31 bit in a few early versions). The fundamental types have a fixed size. This is described in the Delphi help topic "Integer types".

You may be tempted to say that fundamental types are the way to go, since they are well-defined. Usually I would agree. But there is a better way: **use the types that the Windows unit predefines**. A type like *LPARAM* changed from 16 bit to 32 bit (and on Win64, to 64 bit) over the versions of Windows. But most declarations of the 16 bit API didn't change considerably.

Using the types declared in the original header will ensure that your translation is more portable. So use *DWORD* instead of *Cardinal*, *HRESULT* instead of *Longint*, or *LPCSTR* instead of *PChar*, if these are the types in the original declaration. Most of these types will be defined in the Windows unit, or one of the other stock WinAPI units like ShellAPI or MMSystem (or in newer versions, any of the Winapi.*.pas units).

But in case the generic types are used in the original C declaration, here is a short table of corresponding types in C and C++ on one side and Delphi on the other. These values are only valid for current Win32 C and C++ compilers.

C or C++ type	**Delphi types**		**Size**
(Win32)	**signed**	**unsigned**	**(bytes)**
int	Integer	Cardinal	4
long (int) [1]	Longint	Longword	4
short (int) [1]	Smallint	Word	2
char	Shortint	Byte	1
char [2]	AnsiChar		1
wchar_t	WideChar		2
float	Single		4
double	Double		8
long double	Extended		10
__int64 [3]	Int64	UInt64	8
void	*none*		0
void *	Pointer		4
char *	PAnsiChar		4
wchar_t *	PWideChar		4
int *	PInteger		4

[1] the use of the keyword *int* together with *signed, unsigned, long* and *short* is optional. If no type is specified with these keywords, *int* is implied. [2] if used without *signed* or *unsigned*. [3] non-standard extension, but widely used in Win32.

Note: This table applies to Win64 too, except that in Win64, pointers are of course 8 bytes in size.

*The fact that Longword and Longint are **fixed size** types does not hold on some of the newer platforms that Delphi supports, these days. On some platforms, like iOS 64 bit and Linux 64 bit, Longint and Longword are 64 bit types, i.e. they are 8 bytes in size. If you need fixed size types, use the newer aliases like Int32 or UInt32. These are certain to be of the right size.*

There are a few things to note.

- *DWORD* was defined as *Longint* in early versions of Delphi. This could cause some incompatiblities with code that

expected an unsigned type. Later versions correctly define it as *Longword*, but this change can cause incompatiblities between versions. Better check Windows.pas what your version of Delphi uses.
- *LPCTSTR* and *LPTSTR* have a different meaning depending on the definition of the *UNICODE* macro. If it is defined, they mean *LPCWSTR* and *LPWSTR* respectively, otherwise they mean *LPCSTR* and *LPSTR*. More on this in the Unicode section.

This list does not pretend to be complete. There may be more oddities. I will add them as soon as I learn about them

Boolean types

In early C, there was no own type for booleans. Int was simply used for that, and any value that was not 0 was considered to be *true*, and 0 was *false*. Sometimes symbols like *TRUE* and *FALSE* were defined with suitable values. Later on, the type *bool* was introduced, which is equivalent to Delphi's *Boolean*, but this is not often used in headers.

Delphi has the types *ByteBool*, *WordBool* and *LongBool*. If you see types that look like a Boolean being used in a header, with names like *BOOL*, you should either use the same type in your translation, or, if you know the size, use one of the three Delphi types mentioned above. Don't be seduced to use *Boolean* unless the keyword *bool* (case sensitive!) is explicitly used. That is because if the function returns any other value than a 1, the Delphi code will not work properly with *Boolean*, but it will treat the other types in a special way. The three types mentioned above will be converted to a true *Boolean* where this is required in the Delphi code.

Unicode and PAR files

In many headers, you'll see conditional code that is compiled differently if the *UNICODE* macro is defined. An example from *winbase.h*:

```
WINBASEAPI UINT WINAPI GetWindowsDirectoryA(
    OUT LPSTR lpBuffer,
    IN UINT uSize
    );
WINBASEAPI UINT WINAPI GetWindowsDirectoryW(
    OUT LPWSTR lpBuffer,
    IN UINT uSize
    );
#ifdef UNICODE
#define GetWindowsDirectory  GetWindowsDirectoryW
#else
#define GetWindowsDirectory  GetWindowsDirectoryA
#endif // !UNICODE
```

Many API routines and structures come in two flavours, an ANSI and a Wide or Unicode flavour. The ANSI routines usually have PAnsiChar parameters for strings, while the Unicode routines have PWideChar parameters. In some headers, an alias is generated in the way you can see above. Which function is actually defined depends on the *UNICODE* macro. In Delphi, you often see three declarations, as in Windows.pas:

```
function GetWindowsDirectory(lpBuffer: PChar;
  uSize: UINT): UINT; stdcall;
{$EXTERNALSYM GetWindowsDirectory}
function GetWindowsDirectoryA(lpBuffer: PAnsiChar;
  uSize: UINT): UINT; stdcall;
{$EXTERNALSYM GetWindowsDirectoryA}
function GetWindowsDirectoryW(lpBuffer: PWideChar;
  uSize: UINT): UINT; stdcall;
{$EXTERNALSYM GetWindowsDirectoryW}
```

But in the *implementation* section, this is actually resolved as:

```
function GetWindowsDirectory; external kernel32 name
  'GetWindowsDirectoryA';
function GetWindowsDirectoryA; external kernel32 name
  'GetWindowsDirectoryA';
function GetWindowsDirectoryW; external kernel32 name
  'GetWindowsDirectoryW';
```

So *GetWindowsDirectory* is actually an alias for the ANSI equivalent. If you want to use the wide variety, you'll have to use the W name explicitly.

Since Delphi 2009, the change to UnicodeString as the default string type has changed the defualt types for Char and PChar too, to WideChar and PWideChar respectively. Also, the default API functions to be called are now the wide versions (function names ending in `W`, instead of in `A`). Your translations should reflect this too. The stock Winapi units in Delphi already do.

The WPAR utility and PAR files

To make this a little easier and to avoid having to translate both flavours separately, with an increased chance for misalignments and errors, Borland wrote a little command line utility, wpar.exe (this may be temporarily offline, as EDN is being dissolved?). This allows you to only write the declarations once and generate a .pas file containing all versions. This means that you don't write a .pas file yourself, but a file with extension .par, which contains some tags to help the utility create both (or all three) versions, and to indicate where the differences should be.

One example of a struct and a function from *wininet.h* (slightly reformatted, and some comments removed):

```
typedef struct _INTERNET_BUFFERSA {
    DWORD dwStructSize;
    struct _INTERNET_BUFFERSA * Next;
    LPCSTR   lpcszHeader;
    DWORD dwHeadersLength;
    DWORD dwHeadersTotal;
    LPVOID lpvBuffer;
    DWORD dwBufferLength;
    DWORD dwBufferTotal;
    DWORD dwOffsetLow;
    DWORD dwOffsetHigh;
} INTERNET_BUFFERSA, * LPINTERNET_BUFFERSA;
typedef struct _INTERNET_BUFFERSW {
    DWORD dwStructSize;
    struct _INTERNET_BUFFERSW * Next;
    LPCWSTR  lpcszHeader;
    DWORD dwHeadersLength;
    DWORD dwHeadersTotal;
    LPVOID lpvBuffer;
    DWORD dwBufferLength;
    DWORD dwBufferTotal;
    DWORD dwOffsetLow;
    DWORD dwOffsetHigh;
} INTERNET_BUFFERSW, * LPINTERNET_BUFFERSW;
#ifdef UNICODE
typedef INTERNET_BUFFERSW INTERNET_BUFFERS;
typedef LPINTERNET_BUFFERSW LPINTERNET_BUFFERS;
#else
typedef INTERNET_BUFFERSA INTERNET_BUFFERS;
typedef LPINTERNET_BUFFERSA LPINTERNET_BUFFERS;
#endif // UNICODE

//
// prototypes
//

BOOLAPI InternetTimeFromSystemTimeA(
    IN  CONST SYSTEMTIME *pst,  // input GMT time
    IN  DWORD dwRFC,            // RFC format
    OUT LPSTR lpszTime,         // output string buffer
    IN  DWORD cbTime            // output buffer size
    );

BOOLAPI InternetTimeFromSystemTimeW(
    IN  CONST SYSTEMTIME *pst,  // input GMT time
    IN  DWORD dwRFC,            // RFC format
    OUT LPWSTR lpszTime,        // output string buffer
    IN  DWORD cbTime            // output buffer size
    );

#ifdef UNICODE
#define InternetTimeFromSystemTime  InternetTimeFromSystemTimeW
#else
#define InternetTimeFromSystemTime  InternetTimeFromSystemTimeA
#endif // !UNICODE
```

This is coded into the .par file as:

```
type
{#BEGIN}
```

```
  PInternetBuffers{$} = ^TInternetBuffers{$};
{#END}

{#BEGIN}
  {$EXTERNALSYM _INTERNET_BUFFERS{$}}
  _INTERNET_BUFFERS{$} = record
    dwStructSize: DWORD;
    Next: PInternetBuffers;
    lpcszHeader: LPTSTR;
    dwHeadersLength: DWORD;
    dwHeadersTotal: DWORD;
    lpvBuffer: Pointer;
    dwBufferLength: DWORD;
    dwBufferTotal: DWORD;
    dwOffsetLow: DWORD;
    dwOffsetHigh: DWORD;
  end;
{#END}

{#BEGIN}
  {$EXTERNALSYM _INTERNET_BUFFERS{$}}
  INTERNET_BUFFERS{$} = _INTERNET_BUFFERS{$};
{#END}
{#BEGIN}
  TInternetBuffers{$} = _INTERNET_BUFFERS{$};
{#END}

// prototypes
{#BEGIN}
{$EXTERNALSYM InternetTimeFromSystemTime{$}}
function InternetTimeFromSystemTime{$}(const pst: TSystemTime;
  dwRFC: DWORD; lpszTime: LPTSTR; cbTime: DWORD): BOOL; stdcall;
{#END}
```

the `{#BEGIN}` and `{#END}` tags denote one “section” to duplicate. It may require a little playing with this to find out what is actually seen as one “section” by the program. The `{$}` tags are where the A or the W characters should be placed, and the type *LPTSTR* should be used where the different string types should come.

The result of the conversion of the code in the .par file is this .pas code:

```
type
  PInternetBuffersA = ^TInternetBuffersA;
  PInternetBuffersW = ^TInternetBuffersW;
  PInternetBuffers = PInternetBuffersA;
  {$EXTERNALSYM _INTERNET_BUFFERSA}
  _INTERNET_BUFFERSA = record
    dwStructSize: DWORD;
    Next: PInternetBuffers;
    lpcszHeader: PAnsiChar;
    dwHeadersLength: DWORD;
    dwHeadersTotal: DWORD;
    lpvBuffer: Pointer;
    dwBufferLength: DWORD;
    dwBufferTotal: DWORD;
    dwOffsetLow: DWORD;
    dwOffsetHigh: DWORD;
  end;
  {$EXTERNALSYM _INTERNET_BUFFERSW}
  _INTERNET_BUFFERSW = record
    dwStructSize: DWORD;
    Next: PInternetBuffers;
    lpcszHeader: PWideChar;
    dwHeadersLength: DWORD;
    dwHeadersTotal: DWORD;
    lpvBuffer: Pointer;
    dwBufferLength: DWORD;
    dwBufferTotal: DWORD;
    dwOffsetLow: DWORD;
    dwOffsetHigh: DWORD;
  end;
  {$EXTERNALSYM _INTERNET_BUFFERS}
  _INTERNET_BUFFERS = _INTERNET_BUFFERSA;
  {$EXTERNALSYM _INTERNET_BUFFERSA}

  INTERNET_BUFFERSA = _INTERNET_BUFFERSA;
  {$EXTERNALSYM _INTERNET_BUFFERSW}
  INTERNET_BUFFERSW = _INTERNET_BUFFERSW;
  {$EXTERNALSYM INTERNET_BUFFERS}
  INTERNET_BUFFERS = INTERNET_BUFFERSA;
  TInternetBuffersA = _INTERNET_BUFFERSA;
  TInternetBuffersW = _INTERNET_BUFFERSW;
  TInternetBuffers = TInternetBuffersA;

```

```
// prototypes

{$EXTERNALSYM InternetTimeFromSystemTimeA}
function InternetTimeFromSystemTimeA(const pst: TSystemTime;
  dwRFC: DWORD; lpszTime: PAnsiChar; cbTime: DWORD): BOOL; stdcall;
{$EXTERNALSYM InternetTimeFromSystemTimeW}
function InternetTimeFromSystemTimeW(const pst: TSystemTime;
  dwRFC: DWORD; lpszTime: PWideChar; cbTime: DWORD): BOOL; stdcall;
{$EXTERNALSYM InternetTimeFromSystemTime}
function InternetTimeFromSystemTime(const pst: TSystemTime;
  dwRFC: DWORD; lpszTime: PChar; cbTime: DWORD): BOOL; stdcall;
```

The program seems to contain some logic to determine what is actually being converted. That is why *PInternetBuffers* is only aliased to *PInternetBuffersA*, and not declared as *^TInternetBuffers*. The same for *_INTERNET_BUFFERS* and *TInternetBuffers* in the code above. They are aliased to the corresponding ANSI type, and the structure is not repreated once more. The function declaration *InternetTimeFromSystemTime* is completely repeated, however.

Dynamic linking

Sometimes, a certain API function is not available in all versions of Windows. Or different versions of a non-Microsoft API contain different functions. If you have translated a header, it is always a good idea to check several OS versions (or different setups for your DLL) to see if each function is available.

The simplest test is to simply create a console application which uses your unit or units, and writes the address of each function. An example from a recent project (*Shell API* headers):

```
program ShellAPIProject;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  RVSHFolder in 'RVSHFolder.pas',
  RVShLWAPI in 'RVShLWAPI.pas',
  RVShellAPI in 'RVShellAPI.pas',
  RVShlObj in 'RVShlObj.pas',
  RVUrlMon in 'RVUrlMon.pas',
  RVShAppMgr in 'RVShAppMgr.pas',
  RVShlDisp in 'RVShlDisp.pas',
  RVUrlHist in 'RVUrlHist.pas';

begin

  // ShellAPI import test

  Writeln;
  Writeln('shellapi');
  Writeln;

  Writeln(Format('%p -- %s', [@DragQueryFileA, 'DragQueryFileA']));
  Writeln(Format('%p -- %s', [@DragQueryFileW, 'DragQueryFileW']));
  Writeln(Format('%p -- %s', [@DragQueryFile, 'DragQueryFile']));
  Writeln(Format('%p -- %s', [@DragQueryPoint, 'DragQueryPoint']));
  Writeln(Format('%p -- %s', [@DragFinish, 'DragFinish']));
  Writeln(Format('%p -- %s', [@DragAcceptFiles, 'DragAcceptFiles']));

  // Snip hundreds of similar lines

  Readln;
end.
```

I usually use the macro recorder in the Delphi editor to create a file like the above (record the tasks for one line, and then use that recorded macro for each matching line).

The program tries to display the address of each function. If one of the functions is not supported, you will get an error. Where that error occurs is not known yet, since the error will happen before anything is output. If that happens, I usually comment out all functions, and then uncomment lines in big chunks and rerun the program, until the error occurs again. Then I refine the search. That way you can find all functions that are not supported, and if you do it a little strategically, it won’t take a long time.

But what next? I can of course simply remove these functions from the translation and tell people not to use them. But I can also link to the DLL dynamically, and load these functions manually, providing an alternative if they don’t exist.

To do that, I replace the direct function declarations with procedural variables of the same signature (there is no need to predefine a specific type for each function; *var* accepts direct type declarations). So an original:

```
function SHEnableServiceObject(const rclsid: TCLSID;
  fEnable: BOOL): HResult; stdcall;
```

then becomes:

```
var
  SHEnableServiceObject: function(const rclsid: TCLSID;
    fEnable: BOOL): HResult stdcall;
```

Later on in the unit, I implement a function *SafeGetProcAddress*, which uses *GetProcAddress* to find the given API function, and returns the address of that function, or the address of a replacement function, if the function could not be loaded from the DLL. The replacement is also implemented in the unit, and usually just a stub returning a result indicating an error, but sometimes it is a real alternative, using other routines to achieve the same result.

Below is an example of my newest *ShlObj* conversion, using that technique.

```
// Replacement functions

// Replacement for SHEnableServiceObject
function _SHEnableServiceObject(const rclsid: TCLSID;
  fEnable: BOOL): HResult stdcall;
begin
  Result := E_NOTIMPL;
end;

// Replacement for SHGetSetCustromFolderSettingsA
function _SHGetSetFolderCustomSettingsA(
  var pfcs: TSHFolderCustomSettingsA; pszPath: PAnsiChar;
  dwReadWrite: DWORD): HResult; stdcall;
begin
  Result := E_NOTIMPL;
end;

// Etc...
function _SHGetSetFolderCustomSettingsW(
  var pfcs: TSHFolderCustomSettingsW; pszPath: PWideChar;
  dwReadWrite: DWORD): HResult; stdcall;
begin
  Result := E_NOTIMPL;
end;

// Rest of functions removed for clarity...

// Helper function
function SafeGetProcAddress(H: HMODULE; ProcName: PChar;
  DefaultProc: Pointer): TFarProc;
begin
  Result := GetProcAddress(H, ProcName);
  if not Assigned(Result) then
  begin
    Result := DefaultProc;
{$IFDEF SFGPA_DEBUG}
    Writeln(shell32, ': ', ProcName, ' replaced');
{$ENDIF}
  end;
end;

var
  Module: HMODULE;

initialization
{$IFDEF SFGPA_DEBUG}
  Writeln;
  Writeln('ShlObj');
  Writeln;
{$ENDIF}
  Module := LoadLibrary(shell32);
  if Module <> 0 then
  begin
    SHEnableServiceObject := SafeGetProcAddress(Module,
      'SHEnableServiceObject',
      @_SHEnableServiceObject);
    SHGetSetFolderCustomSettingsA := SafeGetProcAddress(Module,
      'SHGetSetFolderCustomSettingsA',
      @_SHGetSetFolderCustomSettingsA);
    SHGetSetFolderCustomSettingsW := SafeGetProcAddress(Module,
      'SHGetSetFolderCustomSettingsW',
      @_SHGetSetFolderCustomSettingsW);
    SHGetSetFolderCustomSettings := SHGetSetFolderCustomSettingsA;

    // Rest of functions removed for clarity...

  end;

finalization
  FreeLibrary(Module);

end.
```

As you can see, I simply assign the address I got for SHGetSetCustomFolderSettingsA to the SHGetSetCustomFolderSettings variable as well. In Unicode-enabled versions of Delphi (Delphi 2009 and newer) you should assign SHGetSetCustomFolderSettingsW instead.

If I run this, with *SFGPA_DEBUG* defined, I get (among output about other files):

```
ShlObj

shell32.dll: SHGetSetFolderCustomSettingsA replaced
shell32.dll: SHStartNetConnectionDialogA replaced
shell32.dll: SHOpenPropSheetA replaced
shell32.dll: SHCreateFileExtractIconA replaced
```

If *SFGPA_DEBUG* is not defined, you will not see any diagnostics.

Of course you can also load all functions in a DLL dynamically. In that case, I would not use the technique described for all functions, just for those for which it is not sure they exist everywhere. Otherwise you can simply use *GetProcAddress*.

Miscellaneous

Sometimes C is a little hard to read. Just look at the declaration of *signal* in the *signal.h* header:

```
void (*signal(int sig, void (*func)(int)))(int);
```

I first thought it was a procedural type, like:

```
type
  signal_f = procedure(param: Integer) cdecl;
  signal = procedure(sig: Integer; func: signal_f) cdecl;
```

But that left me with the question what the `(int)` at the very end was for. It took me quite a bit of reading to realize that this was a normal function, returning a `void (*)(int)`, and the declaration of the normal function *signal* is nested in that return type, i.e. it comes where the `*` is. So the correct translation is:

```
type
  signal_f = procedure(param: Integer) cdecl;

function signal(sig: Integer; func: signal_f): signal_f; cdecl;
```

Such things remind me why I prefer to program in Delphi. `;-)`

Rudy Velthuis

Standard Disclaimer for External Links

Disclaimer and Copyright

Last update: Feb. 20, 2019

Back to top