Rudy's Delphi Corner

DLL dos and don’ts

> *I don’t care if it works on your machine! We are not shipping your machine!*
> — Vidiu Platon

In my perusals on *Stack Overflow* and in the *Embarcadero newsgroups*, and recently also the new English language *en.Delphi-Praxis* and the original German language *Delphi-Praxis*, as well as the *Idera forums*, I have seen lots of examples of people trying to interface with DLLs that were written without any consideration on their use in other languages than the one it was written in. They export functions with language-specific arguments or result types, like C++ objects or Delphi strings, or use calling conventions that can only be found in certain compilers, like Visual C++’s *__fastcall* or Delphi’s *register*.

People using a different language, or even a different version of the same language are unable or hardly able to use such DLLs.

In this article, I try to lay down my experience with DLLs and how I think they should be written, so they can be used from (almost) every language on Windows.

Most of my experience with, well, badly — often stupidly — written DLLs comes from conversions as described in my article Pitfalls of converting and from many questions on Stack Overflow and the Embarcadero Discussion Forums.

*Although this is a Delphi-centric site, most of what I write applies to writing DLLs in **any language**.*

*Note that this is not a tutorial on how you can produce a DLL in Delphi — or in any other language. That is well described in the documentation and I bet there are online tutorials for it as well. This article is merely a discussion of the **dos** and **don’ts** when writing a DLL.*

DLLs

DLLs are, basically, libraries providing a set of functions (or procedures) to be called from a program or another DLL. There are certain limits to what most languages can use. Therefore, if I write a DLL, I follow these rules:

- Limit your types to the types the C language has, or to structs/records of these types. In other words, **do not export data types that are specific to a certain language**, like C++ *templates*, *objects* (eg. *std::string*) or Delphi *objects*, *AnsiStrings*, *UnicodeStrings*, etc.
  C is an exception to this, and can be considered some kind of “common denominator”.
- Do not export data or shared memory. Not all languages can handle those. Just export functions. If you want to expose data, use functions for that.
- **Careful with (certain) return values!** These are handled differently by different compilers.
- **Take care to use the proper calling convention**. Some language-specific calling conventions like Delphi’s default *register* calling convention, or C++ ’s *fastcall and* thiscall calling conventions **cannot be handled by many languages**.
- Be sure to use *and document* **valid structure and array alignment**. If necessary, compare with other compilers and use filler bytes.
- In a DLL, do not allocate data that is passed to the user. The user may not have a proper way to deallocate/free such data. If possible, **let the user allocate the structure** (e.g. a string buffer) **and just fill it in the DLL**.
- **Never let exceptions escape a DLL.** Exceptions are language-specific and other languages can very likely

not handle them.
- If possible, **provide a C header** for the functions and the data types used in the DLL, even if the DLL is not written in C. This will make the DLL usable across a large variety of languages.

I will explain these rules in the following sections.

> ***If you can, use packages, not DLLs***
>
> *In Delphi, it makes no sense to export Delphi-specific types from a DLL, even if you are pretty sure that your DLL will only be used by a program written in Delphi.* There will still be issues like memory management or RTTI differences.
>
> *Rather use packages (.BPL files). These contain a lot of metadata that makes them much more suitable to interface with Delphi, and you can pass around any datatype Delphi knows, without any memory management or RTTI issues. Packages can also export types and variables without the need to declare them and without an extra import unit.*
>
> *In other words: Do not use DLLs if you can use packages.*

Parameters

When interfacing with a DLL, parameters are one of the main problems to deal with.

Types

> *Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*- — Linus Torvalds

The main mistake people make when writing DLLs is to use language-specific types, like templates or classes in C++ or Delphi, or AnsiStrings, strings, dynamic arrays, sets, etc. in Delphi.

I found that, **for a DLL, the *common denominator* for the types used is the language C**. Almost *all* languages on Windows are able to interface with a DLL that only exposes C POD types. POD stands for “plain old data” and more or less defines the scalar types like integers, floating point types, character types and pointers, or compound structures (*structs*, *unions*, arrays) which contain only such data types.

A (non-exhaustive) list of the Windows types that can be safely passed across DLL borders are:

C or C++ type	**Delphi types**		**Size (bytes)**	
(Win32)	**signed**	**unsigned**	**32 bit**	**64 bit**
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	
__int64 [3]	Int64	UInt64	8	
void	*none*		0	

void *	Pointer	4	8
char *	PAnsiChar	4	8
wchar_t *	PWideChar	4	8
int *	PInteger	4	8

[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.

Alternatively, if a fixed size is important, you can use the fixed size integral types, like *UInt8* (*Byte*) or *Int32* (32 bit signed integer), *UInt64*, etc. Generally, the name *IntXX* denotes a signed integer, and the name *UIntXX* an unsigned integer. The *XX* part denotes the number of bits, i.e. 8, 16, 32 or 64. Most other languages will have equivalents for these, often even with the same naming convention.

If you pass around compound types (structs, arrays), pass them around as pointers or references.

Bitfields

If you are a C or C++ programmer: although they are a valid C construct, do not use C bitfields. These are very awkward to use in any other language but C or C++.

Extended

Extended *is an IEEE-754 type, and some compilers are able to handle it, but certainly not all. There could also be alignment issues around them, i.e. one compiler may expect them aligned on 16 bytes, while the other aligns them on 10 bytes (in a* struct, *or in an array). It is probably best to avoid* Extended *parameters (or return values) in DLLs.*

COM Types

Another class of types that can usually be passed across DLL borders are the COM-compatible types, like (*IUnknown*-based) interfaces. Note that in C and C++, interfaces are declared as pointer types to structs, while in Delphi, they are reference types already, so the following Delphi declaration:

```
procedure SetInstanceExplorer(const punk: IUnknown); stdcall;
```

is equivalent to C++'s:

```
void __stdcall SetInstanceExplorer(IUnknown *punk);
```

References

Many languages handle references differently. C strictly uses pointers, and that is, in my opinion, the best way to handle them too, because almost all languages on Windows can deal with pointers (even so called "managed" languages like the .NET languages can deal with them, through marshalling). So instead of doing something like:

```
type
  MyRecord =
    ...
  end;
procedure SomeAPI(... var Rec: MyRecord; ...);
```

You should probably do something like:

```
type
```

```
  PMyRecord = ^MyRecord;
  MyRecord =
    ...
  end;
procedure SomeAPI(... Rec: PMyRecord; ...);
```

Now, Embarcadero (and before, Borland) advise the use of *var*. I am a bit ambivalent about this. The simple code example above, with *GetCurMoniker*, is easier to read with *var*, but using *var* can make translating to and from C a little harder, since in C, you need one extra explicit level of indirection. In other words, if you use *var*, you are not translating as verbatim as possible anymore. Reading documentation, which often shows the C syntax, is also a little harder that way.

I think it is best to use what you prefer.

Arrays

Arrays are best passed around as pointers to the first element.

Because in C and in many other languages, arrays do not have an inherent length that can somehow be retrieved from such a pointer, you should have an extra parameter that passes the length (in elements) of that array.

An example:

```
// DLL function:
procedure DrawGraph(PlotPoints: PInteger; Count: Integer); stdcall;

// Usage:
var
  Measurements: array[0..2999] of Integer;
begin
  // Fill measurements array here, and then draw the graph:
  DrawGraph(@Measurements[0], Length(Measurements));
```

Do not declare such array parameters as *var* parameters. This makes it a lot harder to use pointer arithmetic or to use casts to access the single elements of the array. How this can be done can be found in my conversion article

Callbacks

Sometimes you need callback functions, e.g. to let the user handle each item of an enumeration or get feedback from the user in case of a certain event, etc.

Avoid passing around methods of objects or function pointers or some such. I am aware of the fact that sometimes you need callbacks, but then only declare them as pointers to global functions. So in Delphi, do not have any parameters with types declared as *procedure ... of object*, or *reference to ...*.

Callbacks should follow the rules above, regarding parameter/argument types, return values, calling conventions (e.g. *stdcall*), etc.

Delphi and C++Builder

Delphi and C++Builder DLL exported functions should not have the following types of parameters:

Classes

None of them. They require an infrastructure that other languages (except C++Builder) do not have. But even Delphi and C++Builder programs will have problems using them, because the classes have a different memory manager and also different RTTI. Across different versions, the layout of the classes can be different as well.

Just don't do it.

String types

To be exact: *strings*, *AnsiStrings*, *UnicodeStrings*, *ShortStrings*. The first three are automatically reference counted (ARC) types and other languages cannot use these, nor can they update the reference counts as required. Use *PChar*, *PAnsiChar* and *PWideChar* parameters, respectively.

```
function SetWorkDirectoryWrong(const Dir: string): Boolean; stdcall; // BAD
function SetWorkDirectoryCorrect(Dir: PChar): LongBool; stdcall; // GOOD
```

Dynamic arrays.

This is for the same reasons as for strings. Just give your exported functions normal array parameters, i.e. two parameters, one for the first element and one for the length.

Open array parameters

These are the *array of Bla* kind of **parameters**. Internally, these are passed as **two** parameters, a pointer to the first element and a *High()* value. These are highly specific to Delphi and C++Builder. Rather do what I suggested above: provide two parameters, a pointer to the first element and a length.

```
procedure PrintSalesWrong(const Sales: array of Integer); stdcall; // BAD
procedure PrintSalesCorrect(Sales: PInteger; Count: Integer); stdcall; // GOOD
```

Array of const

These are special cases of open array parameters, in fact *array of TVarRec*. Completely unusable in other languages.

More on open array parameters and arrays of const in my article on that subject.

Method pointers and anonymous methods

While these may be nice as callbacks, they are completely unusable by most other languages. So if your DLL function has one or more of these as parameters, the user will not be able to provide them, which often renders your function unusable. If you need callbacks, declare them as plain procedural/functional types, i.e. without *of object* or *reference to*. They should of course follow all the rules explained in this article. If the callback must reference a certain object, provide for some kind of *UserInfo* parameter, probably at best an untyped pointer. The user can then pass that to the function that takes the callback, and the callback can then pass it back to the user, so the function implementing the callback knows what called it. This is a bit like the *Sender* parameter in event handlers.

The various Microsoft APIs have enough examples of callbacks. Do it the same way.

Sets

Pascal sets are not nearly the same as what most other languages consider a set. Hardly any language will be able to use them. Rather have a parameter of an integral type, of which the bits can be set individually.

```
const
  OptionHighlight =   $0001;
  OptionUseColor  =   $0002;
  OptionLineNumbers = $0004;

procedure SetOptions(NewOptions: Integer); stdcall;

// Usage:
```

```
SetOptions(OptionHighlight or OptionLineNumbers);
```

Note that you can freely use these kinds of parameters inside your DLL and also in your programs. Just don't give your exported *DLL functions such parameters.*

Return values

Generally spoken, the types you should not use as parameters should not be used as return values either.

But there is more: return values are handled quite differently by different compilers, even by compilers of a different vendor but for the same language. Generally, 32 bit simple values like integers or pointers are fine. But how a compound type, like a struct or an array, is returned differs greatly.

Structs

The biggest problem is with *structs*/*records*. How these are returned differs greatly from compiler to compiler.

For instance, some compilers allow you to directly return a struct that is not larger than 64 bits in the register combination `EDX:EAX`. Some may return floating point types in a register, or in the FPU stack. Other compilers don't do this the same way, and if they try to access a member of the struct, they access who-knows-what. This can cause nasal demons. Other compilers insert the return value as a last or first pointer parameter. It seems that, despite detailed calling conventions, such details are not handled uniformly. This means that one compiler may ***not*** receive what it expects to get from the other one.

*The solution is not to **return** anything but simple types, like integers, booleans or bytes. To pass values back to the caller, define a struct type and a pointer type to it. Then let the user pass such a struct as parameter, through a pointer, and you simply fill the values of the struct.*

Pointers

Returning pointers can be a big problem.

- Returning pointers to local variables is *undefined behaviour* in most languages.
- If, from your DLL, you return a pointer to data that you allocated from the heap, then you are passing a pointer to data in the DLL heap. But the heap memory of the DLL is very likely not the same as the heap of the receiving code, so if the caller attempts to free/deallocate such data, this will probably fail, because the memory manager of that code may be completely different and it has no access to the internal structures of the DLL (nor does it know about it).
- Returning a pointer to constant data in the DLL (e.g. a version string or a name) makes the pointer unusable when the DLL gets unloaded.
- Returning a pointer to data passed in by the user is safe. Assume the user passes in a C-style string and you want to return a pointer to the first occurrence of a certain substring. That is not problematic, because the pointer is to valid data managed by the user. Returning *NULL*/*nil* instead of a valid pointer can be a problem (some people prefer to code as null-free as possible), but if it is clearly documented, it is the responsibility of the user to check for it. And it is not entirely clear what else you could return if you did not find the substring.

Generally, instead of returning a pointer to data, you let the user allocate a buffer or struct/record. If this is a buffer or an array, also let them pass in its (maximum) length. Then you fill the struct or array with the data the user is requesting. This way, the user who allocated the data structure can also free it again and memory management is not your (the DLL writer's) concern.

There are ways to allocate memory that the user can free again, if it is taken from a common memory pool, like the memory you can allocate with *CoTaskMemAlloc*, because the user can free it with *CoTaskMemFree* again. This is,

outside COM, not very often done, though. I don't know if there are restrictions or drawbacks to this.

Booleans

Booleans are a bit tricky. In some languages, anything that is not 0 is considered true, while in others, booleans are restricted to the ordinal values 0 and 1 only (and in some to 0 and -1).

To handle this, you should either return a defined value (like the HRESULT type defined by Microsoft) ***instead*** *of a boolean, or use a LongBool. The latter is the Delphi equivalent of a boolean integer, i.e. 0 (zero) is treated as False and every other value as True.*

Floating point types

The same provisos apply as for *structs*/*records*. Rather pass them back to the user as reference parameters.

Calling conventions

Calling conventions govern how parameters are passed and how the stack is cleared after the call. In Win32, there are several calling conventions, and not all are supported by all compilers. In Win64, there is only one.

In Win32, there is one main calling convention you should use: *stdcall* (or *stdcall,* _stdcall, etc. in C or C++). There is seldom a good reason not to use it. The Windows APIs use it too.

In Delphi, the default calling convention is *register*, or as it is called in C++ Builder, *fastcall. One might think that this is compatible with Microsoft's equally named calling convention (i.e.* fastcall), but there are differences. They do not use the same set of registers. So be sure not to forget to declare all your exported functions as *stdcall*. Hardly anyone will be able to use Delphi's *register*.

If you program in C or C++, be sure to use *__stdcall* or one of the macros that mean the same thing, like *STDCALL*, *STDAPI*, *WINAPI* and even *PASCAL*.

> For C and C++ programmers*: I am aware of the fact that some C functions with variable arguments require the cdecl (add some underscores where needed) convention. If you can, avoid exporting such functions, or, if you think you must, provide an alternative too, i.e. one that can pass a pointer to an array and a number of elements.*

> *In Win64, there is only one (user) calling convention. Most compilers still let you declare calling conventions like cdecl or stdcall, but these are generally ignored. Win64 does not allow any other calling convention than the standard one, which is more or less like __fastcall, i.e. parameters are passed as registers, and only if there are more parameters than available registers, the stack is used.*

Name mangling

If you are a C++ programmer, avoid name mangling. It makes importing your functions from any other language/compiler combination than yours terribly hard. The user has to find out the mangled names using a program like Dependency Walker or some DLL dump program and then use these to import. There is no simple relation between the function in the header and the name in the DLL, especially since different C++ compilers could be using different ways to mangle.

Instead, declare your exported functions as *extern "C"*. An example:

```
#ifdef __cplusplus
extern "C" {
#endif
```

```
int __stdcall CloseThing(int thing);
int __stdcall ReadThing(int thing, void* buf, unsigned count);
int __stdcall WriteThing(int thing, void* buf, unsigned count);
long __stdcall SeekThing(int thing, long offset, int whence);

#ifdef __cplusplus
}
#endif
```

Alternatively, you can do:

```
extern "C" int __stdcall CloseThing(int thing);
extern "C" int __stdcall ReadThing(int thing, void* buf, unsigned count);
extern "C" int __stdcall WriteThing(int thing, void* buf, unsigned count);
extern "C" long __stdcall SeekThing(int thing, long offset, int whence);
```

If you really, really want to make things easy for your users, use a .DEF file, as described by Microsoft.

```
EXPORTS
  CloseThing
  ReadThing
  WriteThing
  SeekThing
```

Alignment

If you can, use so called “natural” alignment, i.e. each type is aligned on a multiple of its size. If you have a type that is not 1, 2, 4, 8 or 16 bytes in size, use the next largest of these. So 16 bit words are aligned on a 2 byte boundary, 32 bit DWords are aligned on a 4 byte boundary, etc.

Records should be aligned thus, that the types in it are aligned properly, and the record size is also extended, if necessary with filler bytes, to ensure that they are also properly aligned in an array.

Not all compilers handle this always completely correctly, so be sure to compare the alignment of your types with the *de facto standard*, the MS Visual C++ compiler. Most compilers are compatible with it. If the alignment of your types is not the same as the alignment MSVC++ generates, you should rather add filler bytes to make them align the same way.

In C and C++, you can use the new *alignof()* keyword. In Delphi, you will have to do this manually, for instance using a simple function (using extended RTTI):

```
// http://stackoverflow.com/a/14492362/95954
function GetFieldOffset(ARecordTypeInfo: PTypeInfo;
                        const ARecordFieldName: string): Integer;
var
  Context: TRttiContext;
  Field: TRttiField;
begin
  if (ARecordTypeInfo.Kind <> tkRecord) then
    raise Exception.Create('Not a record type');
  for Field in Context.GetType(ARecordTypeInfo).GetFields do
    if Field.Name = ARecordFieldName then
      Exit(Field.Offset);
  raise Exception.CreateFmt('No such field name: %s', [ARecordFieldName]);
end;

// Used like:
    OffsetBottom := GetFieldOffset(TypeInfo(TRect), 'Bottom');
```

If your version of Delphi does not have extended RTTI yet, you can do it like:

```
    OffsetBottom := NativeInt(@PRect(nil)^.Bottom);
```

The latter is, in my opinion, not as elegant, but quite a lot faster, and it produces the same result.

> *Also take a look at the RTTI function I explain in my article about conversions. It nicely displays the entire layout of a record and shows you how to check sizes.*

Allocating buffers

In short: do not allocate in the DLL what you want to pass to the user of the DLL.

> *In this section, I use the example of returning a string, but the same principles apply to all kinds of data, e.g. byte buffers, records, arrays, etc.*

So — generally — don't return, for instance, a string by allocating the memory and returning the pointer to the string like this:

```
// Don't do this!
function GetSomeFileName(var FileName: PWideChar): Integer; stdcall;
```

That will put the burden of freeing the memory allocated for the string on the user. *But your DLL (even if it is a Delphi DLL to be used by a Delphi program) probably has a different memory manager/allocator than your user, so the user **cannot** safely free that memory.* This is almost certainly the case if the receiving program is written in a different language.

There are a few scenarios you can use instead.

User allocates the buffer

The best scenario is the one used by most API functions. The user must allocate the buffer for the string and pass a pointer to it. Additionally, the user must pass the length of the buffer. So your function should look like this instead:

```
function GetSomeFileName(FileName: PWideChar; Length: Integer): Integer; stdcall;
```

The user allocates the buffer for the filename, and its length. Your DLL function fills this buffer with the file name. If the buffer is too short, you indicate this with an error return value (if possible, return the required length in the returned integer, so the user can try again, or 0 if the function succeeded).

Often, if the string is limited in size, like most path and file names, the user can simply allocate a buffer on the stack and call the DLL function:

```
var
  FileName: array[0..MAX_PATH] of WideChar;
begin
  GetSomeFileName(FileName, MAX_PATH);
```

The user can then keep on using the buffer, or assign it to a string.

> *In my opinion, this is superior to all the other methods, discussed below.*

Providing a Free function

Another scenario is providing an extra function that can free the string. The function will be a DLL function, so it has the right memory manager/allocator to do this, for instance:

```
procedure FreeString(FileName: PWideChar); stdcall;
```

> *The user should know that such a string is to be considered read-only and that he or she should free the string, and how.*

Using CoTaskMemAlloc

You could use an allocator that is provided by the OS, like the one provided by *IMalloc* or the corresponding *CoTaskmemAlloc* function. The user can then free the buffer or string using *CoTaskMemFree*. Your function then looks like the original function (the one commented with "Do not do this!").

But be sure to document this. The user won't know it, and may have to initialize the system with CoInitialize or a similar function. Such strings should also be treated as read-only

There are other memory allocation functions provided by the system. They are discussed on this MSDN page.

Returning constant data

Sometimes the data to be returned does not have to be allocated. It can be a constant in the DLL. Then your function can simply return a pointer to it:

```
function GetVersionString: PWideChar; stdcall;
```

But:

- The user should know that the data is constant, i.e. read-only.
- The data will not be accessible anymore when the DLL is unloaded.

These things must be documented.

Beware!

In all but the first scenario (user allocates), the user is best advised to make a copy of the data to his or her private memory, to avoid data loss and to allow manipulation.

Exceptions

Exceptions are highly language-specific. Even C++ exceptions thrown by a DLL written with a different compiler can probably not be handled by the caller. It gets worse if the caller is written in a different language. A C++ compiler probably won't be able to handle Delphi exceptions, and vice versa. C won't be able to handle either, etc.etc.

Some languages may be able to handle system or "foreign" exceptions, but if you are writing a DLL, don't bet on it.

So, NEVER LET EXCEPTIONS ESCAPE A DLL!

If, in your DLL, you call code that causes an exception, be sure to catch it.

Providing a C header

If you are a C++ and/or C programmer, you won't have much difficulty providing such a header file. You will probably need it yourself anyway. But be sure to restrict the data to C, as described above. Avoid name mangling, avoid language specific types and calling conventions and mind all the other things I mentioned above.

The header should not only declare the functions to be used, but also all structs, enums and constants that apply.

Delphi

In Delphi, this looks like it could be some work, but it isn't. Newer versions of Delphi have project options in the IDE that allow you to generate object files and headers:

> *Project menu → Options → Delphi Compiler → Output – C/C++ → C/C++ output file generation*

The last item is a combobox that allows you to create a number of output files to be consumed by C and/or C++. Just choose an option that includes headers.

Note that the header has the extension *.hpp* and may contain some macros or defines that are specific to Delphi and C++ Builder. Be sure to look through the file and remove such stuff. Also, there will be a *.hpp* file for each unit, not for a DLL. So you may have to combine them into one single header file.

C and C++ usually use *.h* as file extension. You can just rename the file(s) generated.

But even in older versions of Delphi, you can tell the compiler to create C++ headers, it is just not as convenient as in the newer versions. Just specifiy the *-JHPNI* command line option. That will make the Delphi compiler generate, among other files, a *.hpp* file.

Conclusion

I know that, in this article, I may sound a little like a teacher. I apologize for that, but the I learned the principles above the hard way, and if you want to write user-friendly DLLs, you should follow them.

Because they are so important, I will recapitulate them once again:

- Only use simple types or structures of these. C is the common denominator.
- Only export functions.
- Do not return structured types (records, structs). Rather "return" them as arguments to the function.
- Be sure to use natural alignment and document it. Compare with the output of Visual C++, if necessary.
- If possible, avoid allocating data passed to the user. Rather have the user allocate buffers which your function fills.
- **NEVER let exceptions escape a DLL**.
- Provide a C header, if you can. Most non-C languages will be able to translate those.

I hope this article was useful. If you have comments or suggestions, just e-mail me.

Rudy Velthuis

Last update: Apr. 20, 2019

Back to top