

DLL Independent Development

Goal: Build a DLL more "other language friendly" and less IDE or platform dependent.
Fit for mission.

DLL or Package ?



- Packages allow also a faster compilation cause only code concerning the "small" application is compiled with each build. In comparison with a DLL, packages provide an easier approach with forms or method calls cause they are owner of the application so they interact better with other classes. On the other side a DLL is more "language friendly" and less IDE dependent.
- Advantages of Packages:
 - - packages are specific to Delphi
 - - packages are UML conform
 - - faster compilation, less parsing
 - - packages save memory for many applications
 - - testing becomes more planable (DUnit)
 - - better installation, setup's and deployment possible
 - - scalable in product, e.g. a light- or professional version

Versioning Problems ?



- When we update a DLL (change function's implementation), we simply compile it, export some new routines and ship the new version. All the applications using this DLL will still work (unless, of course, you've removed existing exported routines).
- On the other hand, when updating a package, you cannot ship a new version of your package without also updating the executable. This is why we cannot use a unit compiled in D6 in a D7 project unless we have the unit's source; the compiler checks version information of DCU's and decides whether an unit has to be recompiled so any package that you provide for your application must be compiled using the same Delphi version used to compile the application.
- Note: You cannot provide a package written in D6 to be used by an application written in D5.

DLL or Package in OOP



The first decision we should make is:

- Do we put components in a package or do we modularize a whole application with a lot of forms, data-modules, multi-language or resources?
- That means dependencies between classes should be stronger (inner coupling) than dependencies between packages.
- Imagine a scheduling system for public transports with an optional messaging system or an optional reporting system. Some installation need only the schedSystem and others will install the ME and REP module too.

DLL or Package III



The advantage of a DLL is:

For most applications, packages provide greater flexibility and are easier to create than DLLs. However, there are several situations where DLLs would be better suited to your projects than packages:

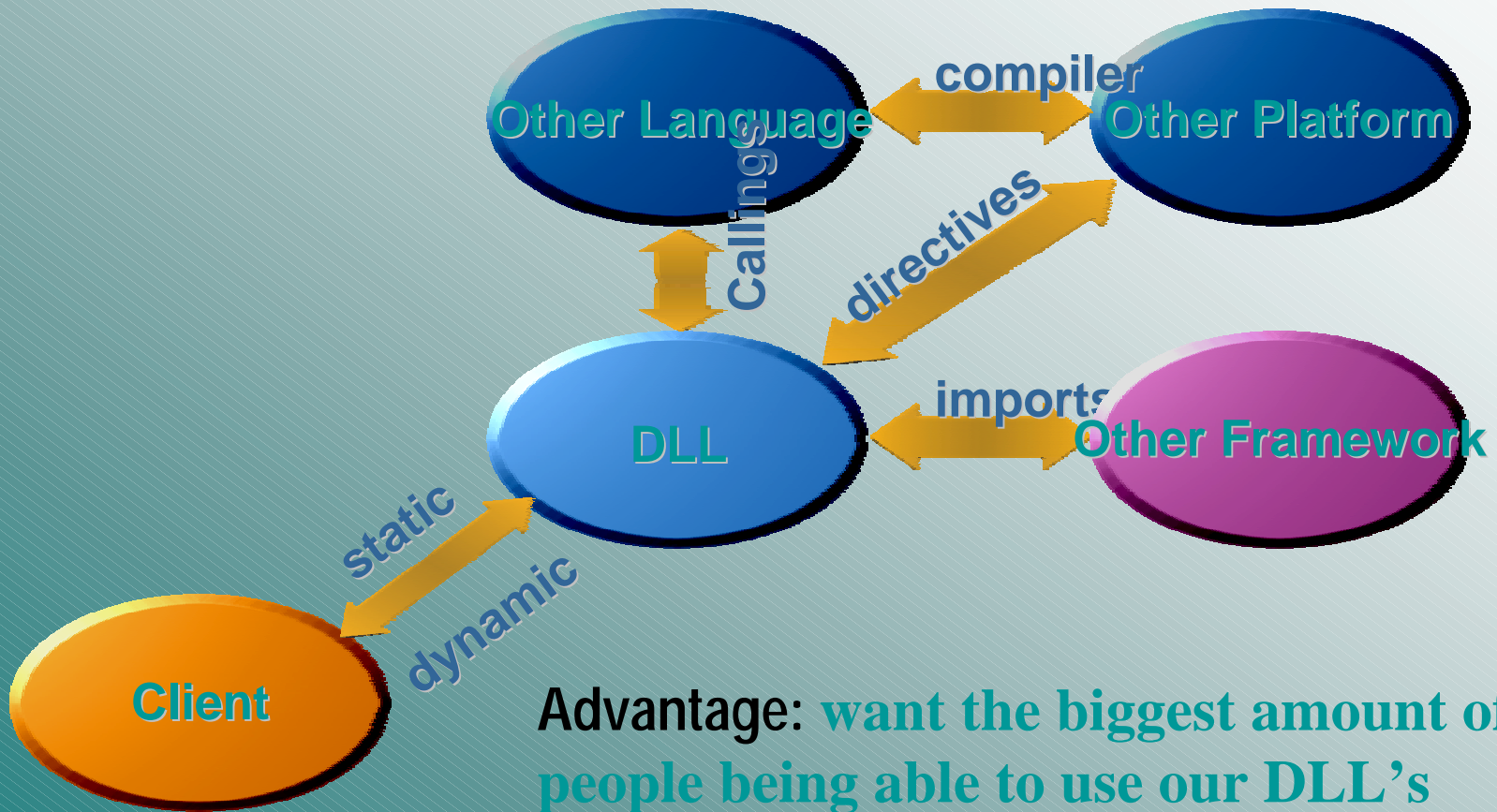
- Your code module will be called from non-Delphi applications.
- You are extending the functionality of a Web server.
- You are creating a code module to be used by third-party developers.
- Your project is an OLE or .Net container.

DLL independencies what's all about?



- Multilanguage
- Memory Management
- Include Files
- Call Convention
- Data alignment and Types
- Name Mangling
- Solutions: DLL+, Call a DLL, Multilang, Tools
 - - to C
 - - to CLX
 - - to .Net

DLL Multi Environment



Advantage: want the biggest amount of people being able to use our DLL's

Memory Management



Delphi uses a mixed memory model, but it is very close to the "C" large model. In a DLL you can use sharemem.pas:

- This seems to be the simplest solution, and requires no special precautions.
- But there's a tradeoff: all allocation calls are diverted into the BorIndmm.dll and may be 2-7 times slower than normal allocation and you need to distribute the dll with your app.

- Code Ex.: a form handle in a DLL (SC_MC.exe)

Include Files



The `$I` parameter directive instructs the compiler to include the named file in the compilation. In effect, the file is inserted in the compiled text right after the `{$I filename}` directive. The default extension for filename is `.pas`. If filename does not specify a directory path, then, in addition to searching for the file in the same directory as the current module, unit recompile if file newer.

To specify a filename that includes a space, surround the file name with single quotation marks: `{$I 'My file'}`.

Ex.: `maXbox pascal script`

Calling Conventions (pascal;)



- When you call the DLL written in C or C++, you have to use the `stdcall` or `cdecl` convention. Otherwise, you will end up in violation troubles and from time to time the application may crash. But for the rest of the world you can use `pascal`! By the way the DLL, you are calling, should be on the search path;).
- So these conventions (`stdcall`, `cdecl`) pass parameters from right to left. With `cdecl`, the caller (that's Delphi) has to remove the parameters from the stack when call returns, others clean-up by the routine.

Tip in c++: `DLL_IMPORT_EXPORT` means after all `stdcall`.

Data types



To understand which operations can be performed on which expressions, we need to distinguish several kinds of compatibility among types and values. These include:

- Type identity
- Type compatibility
- Assignment compatibility

The common type system defines how types are declared, used, and managed in the runtime, and is also an important part of the runtime's support for cross-language integration.

Data alignment



Records or pointer to records should have the right alignment:

- Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest.
- Applications should generally align structure members at addresses that are "natural" for the data type and the processor involved. For example, a 4-byte data member should have an address that is a multiple of four. **#pragma pack**
- Using **packed** in Delphi slows data access and, in the case of a character array, affects type compatibility !



Data alignment II

- *Directive \$A controls the alignment of fields in record types and class structures.*
- *In status {\$A 1} or {\$A-} fields don't get an alignment. All records and structures of classes will be packed.*
- `type`
`TTteststruct = record`
 iVar : Integer; { 4 Byte }
 dVar : double; { 8 Byte }
 bVar : boolean; { 1 Byte }
 *sVar : Array[1..50] of char; { n * 1 Byte }*
`end;`



Type example

To avoid BORLNDMM.DLL in linker pass strings as PChar or ShortString:

- function SelectDir(Dest: PChar): Boolean;
- if dirSelect.ShowModal = idok then begin
- StrPCopy(Dest,
 dirSelect.DirectoryListBox1.Directory);

Caller:

- Var pathback: array[0..MAX_PATH] of Char;
- SelectDir(pathback)

Type example II



In DLL:

```
function TIncomeReal.getPChar: PChar;
```

- begin
- result:= StrAlloc(20);
- StrCopy(result, 'Science not Fiction!');
- end;
- procedure TIncomeReal.freePChar(p: PChar);
- begin
- StrDispose(p);
- end;

In EXE:

- p:= incomeRefN.getPChar;
- memo1.Lines.add(p);
- incomeRefN.freePChar(p);

Types for all



Deliver types others can understand:

C-DLL:

```
bool WINAPI calcBau (HED hInstanz, const char *  
    pszZone, const char * pszBaut, const char * pszHeiz,  
    const char * pszRand);
```

Translation to Pascal:

```
function CalcBau (EHandle: TEidHandle; pZone,  
    pBaut, pHeiz, pRand:PChar): boolean; stdcall;
```

TEidHandle is LongWord, with Word it crashes...

Types for all II



Don't use unknown objects or structs like string or stringlist between different compilers or class libraries in a DLL:

Offsets and alignments are different!

Ex.: C++TStringList → Delphi (mylist.add('hi c++dll'))
→C++TStringList

It won't work to fill the stringlist from c++ DLL

```
TStringList *MyStringList = new TStringList;
```

Import Units



Import Units wraps the functions of the API in DLL's (Delphi → C)

```
const
  {$EXTERNALSYM SC_SCREENSAVE}
  SC_SCREENSAVE = $F140;
```

```
mmsyst = 'winmm.dll';
```

implementation

```
function auxGetDevCaps; external mmsyst name 'auxGetDevCapsA';
function auxGetDevCapsW; external mmsyst name 'auxGetDevCapsW';
function auxGetNumDevs; external mmsyst name 'auxGetNumDevs';
function auxOutMessage; external mmsyst name 'auxOutMessage';
function CloseDriver; external mmsyst name 'CloseDriver';
```

Name Mangling



The cause of diff. problems is the not standardised "name mangling" of different compilers, which decorates the signature of an method to guarantee overloading. So the Vc++ compiler (linker) puts some information about types and parameters on the entry point which the caller doesn't know.

Decorated names were originally created to allow C++ to work with legacy linkers (that might not understand uppercase/lowercase, namespaces, class names, and overloading). In practice these "decorated names" are still around for reasons of compatibility.

Prevent Name Mangling



You can work with an index instead of a name in a .DEF file and export section (depends on your signature)

C++:

```
LIBRARY mxlump_dll
```

```
EXPORTS
```

```
FunctionName1 @1
```

```
FunctionName1 @2
```

```
ProcedureName1 @3
```

Solution: Set an alias in the Delphi external declaration:

```
function CreateIncome2: CIncome; stdcall; external
```

```
'income.dll'
```

```
    name '_CreateIncome';
```



Prevent Name Mangling II

You can work with an block to prevent all functions from decorating:

macro `NoMangle` means `'extern "C"'`

```
extern "C"
```

```
{ __declspec(dllexport) CIncome *CreateIncome();  
  void __EXPORT_TYPE SayHello2();  
}
```

```
function CreateIncome2: CIncome; stdcall; external  
'income.dll'
```

```
  name '_CreateIncome';
```

Prevent Name Mangling III



You can work with an NoMangle decorator macro to prevent the decoration!:

C:

```
NoMangle long DLL_IMPORT_EXPORT  
csp2GetDeviceId(char szDeviceId[8], long nMaxLength);
```

Call it from Pascal:

```
function csp2GetDeviceId(szDeviceId: PChar; nMaxLength: Longint):  
Longint; stdcall; external 'csp2.dll' name 'csp2GetDeviceId';  
var  
    myBuffer: array [0..7] of Char;  
begin  
    csp2GetDeviceId(@myBuffer[0], SizeOf(myBuffer));
```

DLL +



As I stated in an earlier article/session, it's possible to get an object-reference out from a DLL. This technique is known under the name DLL+.

Component Download: <http://max.kleiner.com/download/cplusplus.zip>
Article: <http://max.kleiner.com/dllplus.htm>

initialization

```
MyIntObject:= TMyObject.Create;
```

finalization

```
MyIntObject.Free;
```

- But how about the DLL is written in C++?

DLL +



First of all, you have to translate the header-file (should be delivered with the DLL), which is like an interface-section in OP. Headers in c usually contain all sorts of definitions which are relevant outside the module. In our c++ example it looks like:

- `/*FILE: income.h */`
- `class CIncome`
- `{`
- `public:`
- `virtual double __stdcall GetIncome(double aNetto) = 0 ;`
- `virtual void __stdcall SetRate(int aPercent, int aYear) = 0 ;`
- `virtual void __stdcall FreeObject() = 0 ;`
- `};`

DLL +



Then you translate it to an Abstract Class in a own unit:

- //FILE: income.pas
- interface
- type
- CIncome = class
- public
- function GetIncome(const aNetto: double): double;
- virtual; stdcall; abstract;
- procedure SetRate(const aPercent: Integer; aYear: integer);
- virtual; stdcall; abstract;
- procedure FreeObject; virtual; stdcall; abstract;
- end;

DLL +



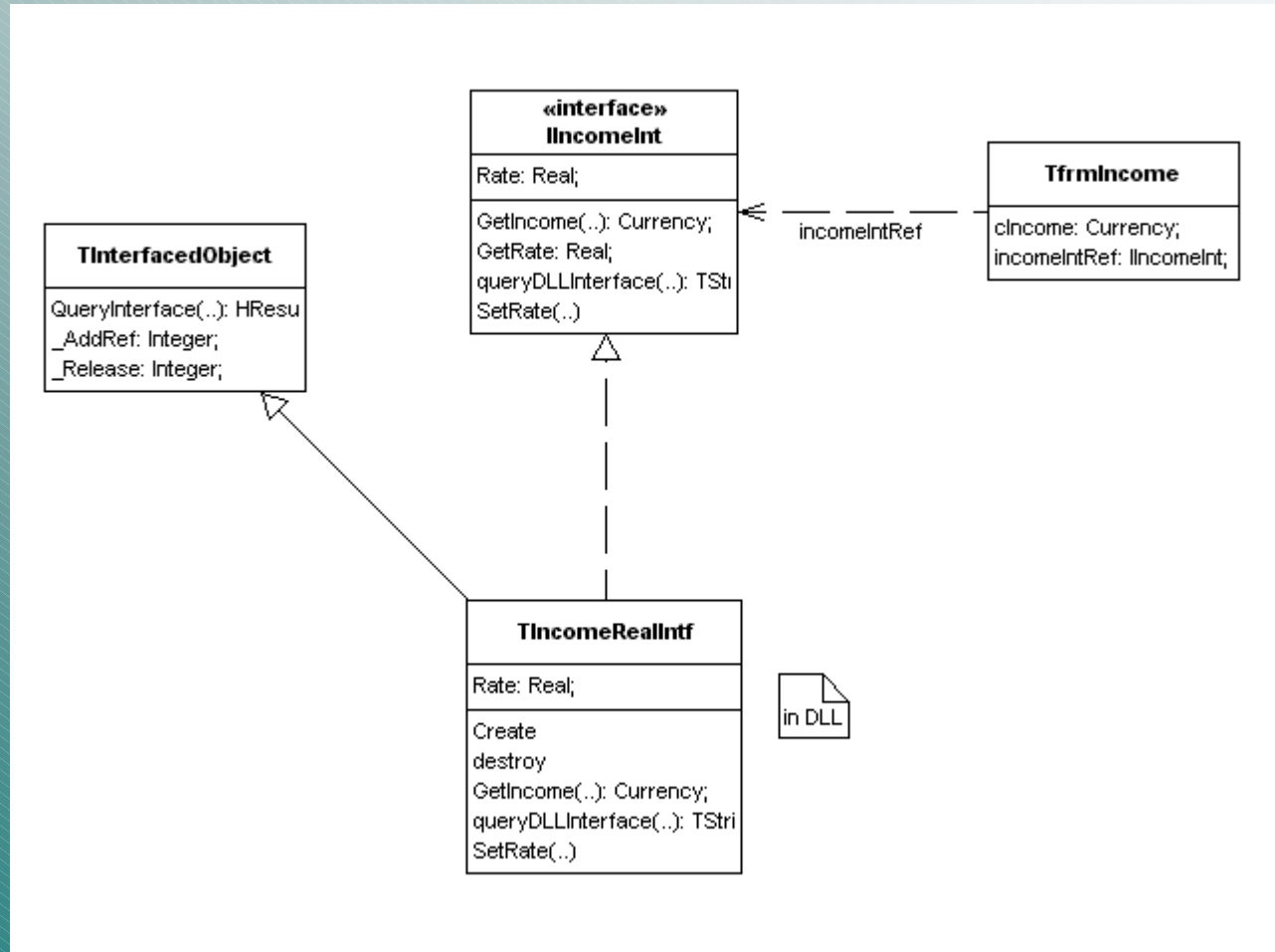
In the c++ DLL, there is a procedure FreeObject, this is necessary because of differences in memory management between C++ and OP:

- `void __stdcall FreeObject()`
- `{`
- `delete this ;`
- `}`

- At least the DLL-call is simple:

- `incomeRef: CIncome; //member of the reference`
- `function CreateIncome: IIncome;`
- `stdcall; external 'income_c.dll';`

DLL+ Interface - Implementation



Ressourcen DLL as MultilangTranslator Unit



- langfile2.rc
- STRINGTABLE
- { 3, "Arbeiten im EKON Team"
- 4, "Inhalt"
- 1003, "work in team1"
- 1004, "Content"
- library reslang2;
- {\$R 'langfile2.res'}
- begin
- end.
- http://max.kleiner.com/download/multilang_intro.pdf

Use a .net assembly from Delphi 7



When we import functions with stdcall in Delphi 7 from the Delphi.net-Assembly unmanaged exported procedures, the call should be a success:

Delphi 9: .NET Unmanaged Export

```
library D9LibExport;  
{ UNSAFECODE ON}  
uses  
  SysUtils,  
  Classes,  
  System.Reflection;  
exports  
  D9NET_UserName, D9NET_UserNameEx;
```

Use a C# assembly from Delphi 7

Mit COM Interop



Die C#-Implementierung wird als COM-Objekt verpackt, so dass Delphi 7 über den COM-Interop-Weg (RegAsm.exe CSharpObj.dll /tlb:CSharpObj.tlb) darauf zugreifen kann, wenn erst einmal die von RegAsm generierte Typbibliothek (*.TLB) vorliegt:

```
using System.Reflection;
using System.Runtime.CompilerServices;
//
using System.Runtime.InteropServices;
...
[assembly: ComVisible(true)]
[assembly: ClassInterface(ClassInterfaceType.AutoDual)]
```

CLX DLL



- **Kylix** supports shared objects (.so's) which are the equivalent of **.dll's**
But you can't use dll's **AND** .so in **Kylix**.
- You need to understand that a Windows **dll** has the Windows PE file format. A shared object under Linux has the ELF file format.
- One option is to use a conditional compile based on the operating system.
Use `cdecl` on Linux and `stdcall` on win32 (**Kylix** makes this easier; if you declare `stdcall`, it will be compiled as `cdecl`!)
- `procedure InitC; cdecl;`
- `procedure InitS; stdcall;`

Call from Client platform independent



```
begin
  {$IFDEF LINUX}
    dllhandle:= dlopen(PChar(s2), RTLD_LAZY);
  {$ELSE}
    dllhandle:= LoadLibrary(Pchar(s2));
  {$ENDIF}
  if dllhandle = {$IFDEF LINUX} NIL {$ELSE} 0 {$ENDIF} then

  {$IFDEF LINUX}
    p.Ext1:= dlsym(dllhandle, pchar(copy(s, 1, pos(#0, s)-1)));
  {$ELSE}
    p.Ext1:= GetProcAddress(dllhandle, pchar(copy(s, 1, pos(#0, s)-1)));
  {$ENDIF}
```

DLL Tools on board



- DCC32 compiler with JHPNE as option will generate C++ headers (with .hpp extension) for your units!

```
DCC32 -JHPHN -N D:\DEV\PPT -O D:\DEV\PPT -U  
D:\COMPONENTS\SC2_2\securePtBase.pas
```

- rundll32 income.dll '_SayHello2' //for short tests
- Dependency Viewer shows the inside of dll's:

<http://delphi-jedi.org/Jedi:CODELIBJCL>

- Unmanaged Export on Delphi 2005
library D9LibExport;
{UNSAFECODE ON}



Questions and hope answers ?

max@kleiner.com

