Title: Extending Delphi 95 with DLLs

5/7/95
Type: Technical Session
Track: Delphi 95

Short Description:

Users can customize Delphi using DLLs. This technical session will show how to use both Delphi and C++ to rapidly develop Delphi extension DLLs.

Abstract:

Although Delphi supports a broad range of features, there are always going to be users that want something more. Borland designed Delphi to be extensible through the use of custom controls and DLLs. This presentation will show how to use Delphi and C++ to create user extension DLLs in an object-oriented manner. Users will need to be familiar with Windows program development, C++ and Delphi Object Pascal.

Extending Delphi 95 with DLLs

Author: Ted Faison, Faison Computing

Overview

Delphi is a powerful application development program. It supports not only the development of form-based applications (as in Visual Basic), but also generic Windows applications, capable of supporting even the Doc/View model. To support such generic applications Delphi was designed to be fully extensible, using Delphi controls, VBX controls and regular DLLs. With DLLs, programmers have their choice of development language, allowing them to use standard languages such as C++. Delphi itself can be used to create Object Pascal DLLs that can in turn be used by other Delphi applications, or even by C++ apps. This paper will show how to mix and match Delphi DLLs with C++ and viceversa. OWL programmers will greatly benefit from the use of Delphi DLLs, because they will gain access to the powerful Visual Components used in Delphi forms.

Delphi DLLs

Almost any code you write in a Delphi application can be put into a DLL. Using DLLs may or not be the best approach, depending on the situation, but DLLs do have their advantages: Code developed for one app can be sealed off in a separate DLL for other projects to reuse. Having the code in a separately built module, the interfaces are completely defined and usually stable, hiding implementation internals from callers. Also, if you need to run multiple applications that use the same code, only one copy of the DLL gets loaded into memory. Microsoft touts this as a great feature, but in practice it is rare to see applications sharing the same user DLL, except when you allow multiple instances of the same application to run at the same time.

Creating the DLL

The way you develop Delphi code for DLLs is essentially the same as for a normal EXE project. The two main differences are the following:

- The Project Source Unit is slightly different: The keyword `program` is changed to `library`, there is no executable code and there is an `exports` section.
-
- The types of variables that appear in exported functions should contain Windows-compatible types. For example, the Object Pascal types `string` and `Boolean` should be replaced by `PChar` and `WordBool` respectively.

It is common to create DLLs that include Delphi VCL controls like notebooks, data entry controls, string-grid and user controls, because you can then harness the Delphi power from applications written in languages like C++. Consider creating a data-entry form containing a notebook with tabbed pages. I created such a form and put it in the file INFO.DLL. It can be used to get information about fictitious customers. The form has two pages, shown in Figure 1 and Figure 2.

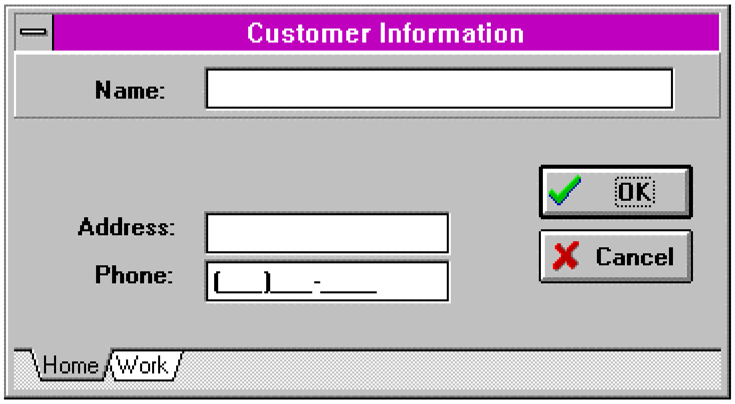

Figure 1 - The first page of the form in INFO.DLL.

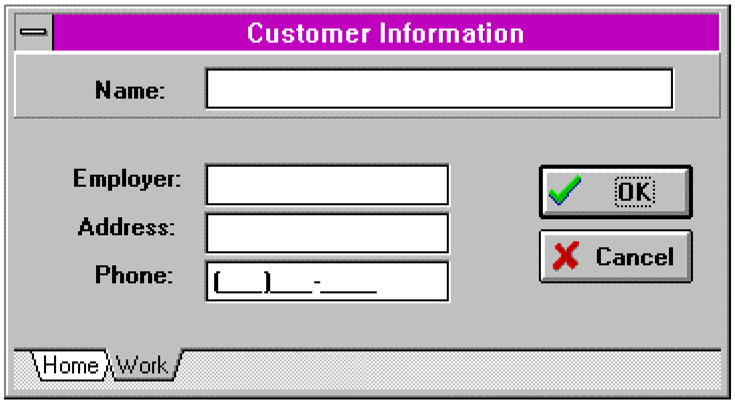

Figure 2 - The second page of the form in INFO.DLL.

Creating such a form is straightforward in Delphi. You start by creating a standard EXE application, so you can test the code immediately. When you're done testing, you convert the project to DLL. You then need to export one or more functions to connect the internal Object Pascal code to your application. I used a function declared in INFO.DLL like this:

```
function GetCustomerInfo(Name: PChar;
                         HomeAddress: PChar;
                         HomePhone: PChar;
                         Employer: PChar;
                         WorkAddress: PChar;
                         WorkPhone: PChar;
                         StringLengths: Integer): WordBool; export;
```

Note the `export` keyword at the end of the declaration. The function takes a series of `PChar` (not Object Pascal `string`) parameters, and fills them with data entered into the DLL's **Customer Information** Form. The size of the `PChar` arrays is passed in the parameter `StringLengths`. The value returned by the function is `True` if the dialog is closed by clicking the OK button is pressed, and `False` otherwise. The code for `GetCustomerInfo` is shown in Listing 1.

```
function GetCustomerInfo(Name: PChar;
                         HomeAddress: PChar;
                         HomePhone: PChar;
                         Employer: PChar;
                         WorkAddress: PChar;
                         WorkPhone: PChar;
                         StringLengths: Integer): WordBool;
begin
  Result:= False;
  CustomerInfoForm:= TCustomerInfoForm.Create(Application);
  try
    if CustomerInfoForm.ShowModal = IDOK then
      begin
        StrPLCopy(Name, CustomerInfoForm.NameEdit.Text,
                  StringLengths);
        StrPLCopy(HomeAddress,
                  CustomerInfoForm.HomeAddressEdit.Text,
                  StringLengths);
        StrPLCopy(HomePhone, CustomerInfoForm.HomePhoneEdit.Text,
                  StringLengths);
        StrPLCopy(Employer, CustomerInfoForm.EmployerEdit.Text, StringLengths);
        StrPLCopy(WorkAddress, CustomerInfoForm.WorkAddressEdit.Text,
                  StringLengths);
        StrPLCopy(WorkPhone, CustomerInfoForm.WorkPhoneEdit.Text, StringLengths);
        Result:= True;
      end;
  finally
     CustomerInfoForm.Free;
 end;
end;
```

Listing 1 - The function exported from the DLL.

The function converts the data entered in the edit boxes into `PChars`, returning the results in the character arrays supplied by the calling application. The function displays the **Customer Information** form by creating a `TCustomerInfoForm` object, created using standard Delphi VCL controls.

Calling the DLL from a Delphi application

Delphi applications that call functions in DLLs are developed no differently than regular Delphi apps. All you need to add is a declaration somewhere in your code to tell the Object Pascal compiler what DLL functions you are going to invoke, and what DLLs they are contained in. To use the function `GetCustomerInfo`, shown in Listing 1, you would add the following declarations to your application:

```
function GetCustomerInfo(Name: PChar;
                         HomeAddress: PChar;
                         HomePhone: PChar;
                         Employer: PChar;
                         WorkAddress: PChar;
                         WorkPhone: PChar;
                         StringLengths: Integer): WordBool; far;
external '..\INFO';
```

Notice two things in particular: the `far` keyword at the end of the function declaration, and the `external` keyword. The filename after `external` has no suffix, although the file is a DLL. If you add the `.DLL` suffix, the compiler will add an additional `.DLL` suffix, and fail to locate the file. If the DLL is not in the path used by Windows to locate DLLs, then you must enter the path with the filename.

To test INFO.DLL I created a small Delphi app called CALLINFO.EXE that has a simple main form, as shown in Figure 3.

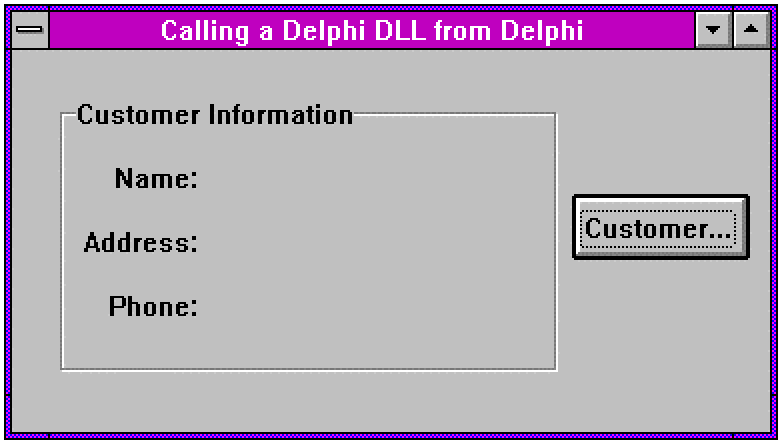

Figure 3 - The main form in CALLINFO.EXE

Clicking the **Customer...** button invokes the `GetCustomerInfo` function in INFO.DLL, using the code:

```
procedure TMainForm.CustomerClick(Sender: TObject);

const
  StringLengths = 255;

var
  Name:        array[0..StringLengths] of Char;
  HomeAddress: array[0..StringLengths] of Char;
  HomePhone:   array[0..StringLengths] of Char;
  Employer:    array[0..StringLengths] of Char;
  WorkAddress: array[0..StringLengths] of Char;
  WorkPhone:   array[0..StringLengths] of Char;

begin
  if (GetCustomerInfo(Name, HomeAddress, HomePhone,
                      Employer, WorkAddress, WorkPhone,
                      StringLengths) = True) then
    begin
       CustomerName.Caption:=    StrPas(Name);
       CustomerAddress.Caption:= StrPas(HomeAddress);
       CustomerPhone.Caption:=   StrPas(HomePhone);
    end;
end;
```

Listing 2 - The code in CALLINFO.EXE that invokes a DLL function.

The procedure fills in the `TLabel` fields in the form if the **Customer Information** Form was closed with the OK button. The `PChars` returned by the DLL function are converted to Object Pascal `strings` using the conversion function `StrPas` before being passed to the `TLabel` objects. Pretty straightforward.

Calling a Delphi DLL from an OWL application

You can call functions exported from Delphi DLLs from an OWL application, but you must keep one essential detail in mind: Delphi functions are `pascal` functions, therefore they must be declared as such in any C++ code that uses them. Consider a simple OWL app whose main window is a dialog box like the one shown in Figure 3. I created an app using AppExpert and called it TESTINFO. The app has a handler for the **Customer** button that calls the Delphi `GetCustomerInfo()` function. The handler code looks like this:

```
// declare the Delphi DLL function before using it
extern "C" {
  BOOL pascal GetCustomerInfo(LPSTR, LPSTR, LPSTR,
                              LPSTR, LPSTR, LPSTR, int);
}

void TDialogTestInfo::CustomerButtonClicked ()
{
  const int SIZE = 255;
  char customerName [SIZE];
  char homeAddress [SIZE];
  char homePhone [SIZE];
  char employer [SIZE];
  char workAddress [SIZE];
  char workPhone [SIZE];

  // call the Delphi DLL function
  if (GetCustomerInfo(customerName, homeAddress, homePhone,
                      employer, workAddress, workPhone,
                      SIZE) != TRUE) return;

  // use the data obtained from the Delphi DLL function
  name->SetText(customerName);
  address->SetText(homeAddress);
  phone->SetText(homePhone);

}
```

Listing 3 - An OWL function that calls a Delphi DLL function.

Pay special attention to the declaration of the Delphi DLL function. The declaration uses two essential items:

- the `extern "C"` declaration
- the `pascal` modifier

The `extern "C"` prevents the compiler from using a mangled name for `GetClassInfo(),` to be resolved by the linker. Name mangling is a C++ feature which allows the same function name to be used for different functions, as long as each function takes a different argument list. For example the function name `print` might be used by the functions `print(int)` and `print(char*).` The compiler mangles the function name by appending special characters that encode the type of parameters the function takes. The resulting function names are later resolved by the C++ linker. Object Pascal knows nothing about C++ name mangling, and wouldn't know how to handle mangled names. Function names exported from Delphi DLLs are never mangled.

The `pascal` keyword is required, since the C++ code is calling an Object Pascal function. Does it matter? Yes, because `pascal` functions differ from C++ functions in two important ways:

- they expect parameters to be pushed on the stack from left to right
- they clean up the stack before returning

C++ functions expect parameters to be pushed on the stack from right to left, and do not clean up the stack before returning (the caller does the clean up). Forgetting the `pascal` modifier will result in a runtime crash when the DLL function returns, because the stack will be cleaned up twice: once by the called `pascal` function, and one by the calling function. The code may even crash before the function returns, depending on what the called function does with the reverse-ordered arguments. An Object Pascal function can be declared using the `cdecl` directive, making it look like a C function to its callers. You might want to declare the functions exported by a Delphi DLL with the `cdecl` directive if you plan to use them from C or C++ programs.

To link the OWL app with a Delphi DLL, you must include the DLL's import `.LIB` file in the project. Delphi doesn't create `.LIB` files when generating DLLs, so you need to create the file yourself, by running the IMPLIB utility provided with Borland C++. For a Delphi DLL named MYFILE.DLL, you would run IMPLIB with the following command line:

```
IMPLIB MYFILE.DLL MYFILE.LIB
```

After linking the project, you're ready to run it. As with any DLL, make sure INFO.DLL is located somewhere in the Windows DLL search path. Generally you'll want DLLs to either be in the working directory, or in the Windows SYSTEM directory. Running TESTINFO, you'll see that it behaves just like the Delphi program that I called CALLINFO.EXE. With all the Delphi controls that are available, it can really give OWL programs a serious boost to be able to use Delphi code through DLLs.

C++ DLLs

OK. So you can use Delphi DLLs with Delphi apps and C++ apps. How about the opposite? Can you use C++ DLLs with Delphi applications? Of course. Would you ever want to? Maybe. If you have code that you have already created in C++, it makes sense to reuse it from Delphi. It the code hasn't been developed yet, it might make sense to just go ahead and write it in Object Pascal. Object Pascal code executes just as efficiently as C++ code. To show how to use C++ DLLs from Delphi. I'll create a sample DLL that does something that would be more difficult to do in Object Pascal **--** justifying the use of C++. Let's talk containers. With the word *containers* I don't mean OLE container applications, but containers in the sense of *collections*, like linked lists and arrays. Borland C++ has lots of container types, with great iterators, making C++ a better implementation vehicle in this case than Object Pascal.

Assume you need a container to store the names and telephone numbers of your customers. If the number of names is small, it may not be worth the trouble to create a full-fledged database application. I created a small C++ DLL called NAMES.DLL that stores names and phone numbers in a dictionary container. Dictionaries use hash tables to store associations, each having a key and a value. In NAMES, the keys are names and the values are the phone numbers, stored as char arrays. NAMES exports a function declared like this:

```
extern "C" {
  BOOL far pascal LookupNumber(char far* theName, char far* theNumber);
```

The function takes a name and looks up the person's telephone number. TRUE is returned if the number was found, otherwise FALSE. There are a couple of important things to note in the above declaration:

1. The `pascal` keyword. By default, Delphi always assumes the functions it calls use the `pascal` calling sequence.

2. The `extern "C"` declaration. By default, C++ exports mangled function names, which are incompatible with Delphi or other languages. Declaring functions `extern "C"` exports their names as unmangled.

The code for the DLL is shown in Listing 4.

```
#include <owl\owlpch.h>
#pragma hdrstop

#include "names.h"

const int ENTRIES = 3;
```

```
static DirectoryListing numbers [ENTRIES] = {
    DirectoryListing("Tom",   "555-1010"),
    DirectoryListing("Dick",  "555-2020"),
    DirectoryListing("Harry", "555-3030")
};

Directory phoneDirectory;

// given a name, lookup the person's telephone number
BOOL far _export pascal LookupNumber(char far* theName, char far* theNumber)
{
  strcpy(theNumber, "<none>");

  if (!strlen(theName) ) {
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(NULL, "Enter a name first.",
               "No name to lookup", MB_OK);
    return FALSE;
  }

  // lookup the name
  DirectoryListing* listing =
    phoneDirectory.Find(DirectoryListing(theName, (char*) 0) );

  if (!listing) {
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(NULL, "Must be one of {Tom, Dick, Harry}",
               "Name not found", MB_OK);
    return FALSE;
  }

  strcpy(theNumber, listing->Value().c_str() );
  return TRUE;
}

int CALLBACK LibMain(HINSTANCE, WORD, WORD, LPSTR)
{
    // initialize the phone directory
       for (int i = 0; i < ENTRIES; i++)
      phoneDirectory.Add(numbers [i]);

    return 1;
}
```

Listing 4 - The OWL DLL code to look up telephone numbers.

The DLL initializes the `phoneDirectory` dictionary with names and numbers in the `LibMain` function. The function `LookupNumber()` receives a name and looks up the corresponding phone number in the dictionary. The caller can test the return value to see if a phone number was found or not.

Calling the DLL from Delphi

I created a small Delphi app called TESTNAME to call the C++ DLL. The application displays the form shown in Figure 4.

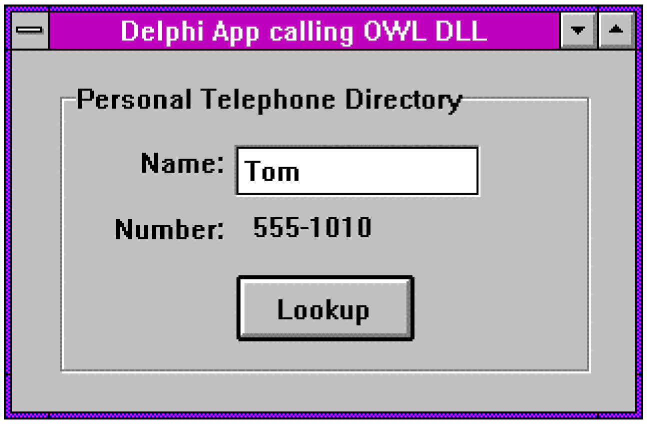

Figure 4 - The form used in the Delphi app to call an OWL DLL.

To make the DLL work with Delphi, you don't need a `.LIB` import library for the DLL, although Borland C++ creates one automatically after building DLLs. Before using functions contained in a DLL, you must declare them. In TESTNAME, the function `LookupNumber` is declared in the `implementation` section, before your code, like this:

```
implementation

{$R *.DFM}

{ declare the function imported from the OWL DLL}
function LookupNumber(theName: PChar; theNumber: PChar): WordBool; far;
external '..\NAMES';
```

TESTNAMES calls the DLL function in the handler for the **Lookup** button in Figure 4, as shown in Listing 5.

```
procedure TForm1.LookupButtonClick(Sender: TObject);

const
  StringLengths = 255;

var
  Name:   array[0..StringLengths] of Char;
  Number: array[0..StringLengths] of Char;

begin
  {get the name to lookup}
  StrPCopy(Name, nameEdit.Text);

  {lookup the person's phone number}
  if (LookupNumber(Name, Number) = True) then

    {show number on the form}
    phoneNumber.Caption:= StrPas(Number);
end;
```

Listing 5 - The Object Pascal code that calls a C++ function in a DLL.

If you declare a function that isn't in the DLL you specify, Delphi doesn't find out until runtime, when it tries to call the function. You'll get the error message:

```
Call to Undefined DynaLink
```

Another mistake you might make is to declare a DLL function to be in a DLL that can’t be found when you run a Delphi program. The error message

```
Cannot find <XYZ.DLL>
```

will appear when you try to run the Delphi program. The DLL must be somewhere in the Windows search path for DLLs.

Summary

Delphi was created to be extended, both by the use of custom VCL controls, and by the use of DLLs. Writing Delphi applications that call DLL functions developed with Delphi or C++ is fairly easy, as long as you keep a couple of things in mind:

1. Use argument types that are Windows compatible. Don't use `pascal`-specific types like `string`, `Boolean`, `ByteBool`, etc. Stick to standard types like `PChar` (equivalent to `LPSTR`) and `WordBool` (equivalent to `BOOL`).

2. When calling a Delphi DLL function from C++, remember to declare the function `pascal` in your C++ code.

3. When creating a C++ DLL function to be called by Delphi, also use the `pascal` modifier, and turn off name-mangling on exported names using `extern "C"`.

Contact Information

Ted Faison can be reached on CompuServe at 76350,1013.