# Writing Cross-Platform Libraries in Pascal

by Kevin Killion
kevin@shsmedia.com

*It is possible and practical to build Pascal units that are usable from both CodeWarrior on the Mac and Delphi on Windows. As experience in Delphi increases, the Mac Pascal programmer will also discover much that is appealing "over there" , more than enough to compensate for quirks and omissions. The live presentation of this paper will be augmented with a demonstration of the development environment described, and running examples.*

## Introduction

The objective is to create Mac and Windows programs in Pascal, using as much common code as possible. The strategy is to use a single code base in Metrowerks CodeWarrior to create non-interface libraries for Mac and Windows. The user interface for Windows will be created in Inprise Borland Delphi (where such things are easy) making calls to the DLL produced by CodeWarrior as needed. The user interface for Mac takes more effort, since no real RAD compiler environments exist. However, in this paper we are coming from the point of view of an experienced Mac developer and so creating the Mac interface is not considered a serious obstacle.

The topics to be covered in this paper include:

- Differences in variable types between the development platforms
- Differences in calling conventions between the development platforms
- Handshaking from apps to DLLs
- Notes about using Delphi, from a Mac lover's point of view

The live presentation at MacHack will illustrate these issues with a working crossplatform project, live examples of both environments, and some operational issues.

## About Delphi

Delphi is a RAD (rapid application development) IDE that sports tightly integrated interface construction and program development, all based on a variant of Pascal. For those unfamiliar with RAD programming, there is simply no product of equivalent power and price on the Mac, sad to say. The closest approximation may be Visual Café , which does not do native compilation, is not nearly as ambitious in scope, and is not as well supported by third-party products.

Delphi is a product of Inprise Corporation, formerly known as Borland. The Borland name survives as a brand name for Inprise's family of development products.

## How to Create a DLL with CodeWarrior Pascal

Surprise! This step is easy: Metrowerks provides project stationery to create a new Windows DLL using Pascal. You'll find the sample code (dll.p) to be beautifully simple.

Using CodeWarrior's wondrous ability to specify multiple targets in one project, it's possible to generate a Mac 68K application, a Mac PowerPC application and a Windows DLL within a single Pascal project, making use of the same core Unit for all targets. This has been done in the "Mac / Win / DLL" project which is supplied with this paper.

Our focus here is on developing libraries for non-interface purposes. However, it's helpful to have a supplemental unit of routines to implement crucial platform-specific functions. An example is declaring, accessing and disposing memory blocks, which we implement in "PortableMemory.p". This will be described in the live session.

## Passing Data Between CW and Delphi

A key challenge is understanding differences between how CodeWarrior and Delphi store and handle various data types. We'll cover that in this section.

### Numeric Types

Watch out for differences in numeric types. When passing values between a DLL and a host application, you need to make sure that both sides are using the correct data type.

For example, a "real" is defined as a 4-byte floating point value on Think Pascal and Metrowerks, but "real" is defined as a 6-byte float on Delphi. To pass a 4-byte float, you need to use the type "single" on Delphi.

You must also be careful with integers: the type "integer" has different lengths in different versions of Delphi. In 16-bit Delphi, "integer" is two bytes, in 32-bit Delphi, "integer" is four bytes.

Here are some equivalents:

**Floating Point Numerics**

| #bytes | Mac | Delphi |
|---|---|---|
| 4 | real | single |
| 6 | n.a. | real |
| 8 | double | double |

**Integers**

| #bytes | Mac Think Pascal | Mac CW Pascal | Delphi |
|---|---|---|---|
| 1 | n.a. | n.a. | shortint |
| 2 | integer | integer | smallint |
| 4 | longint | longint | longint |
| 2 or 4 | -- | -- | integer |
| 8 | computational | comp | comp |

Note that byte length is only one characteristic to determine if types match, especially with the floating point types. The types shown in the tables above do match on all characteristics; for example, a Mac "real" and a Delphi "single" have the same internal representations.

Note that Think Pascal's "computational" type is stored just as you'd expect an integral value to be stored, and that's identical to CodeWarrior's and Delphi's "comp" type. Nonetheless, Think Pascal prefers to consider "computational" as a special type of real, that has no decimal places. Go figure.

### Char Type

I ran into problems passing a char as a parameter between Delphi and a CodeWarrior DLL. Consider this bit of code in Delphi:

```
type
  k: char;
  num: smallint;
begin
  k := 'X';
  DoSomething(k,num);  { pass "k" to the
DLL}
end;
```

Here is the routine in the DLL:

```
procedure DoSomething(k:char; var
num:integer);
begin
  num := ORD(k);
end;
```

I expected that num would now equal 88, the ASCII value for “X”. Instead, in one case, the ordinal value of the character was -168.

It seems to me that when sent by Delphi to the DLL, the lower order byte of the variable for the character is correct, but the higher byte is undefined. The Delphi language guide says, “A Char is stored as an unsigned byte”, and this is subtly different from CodeWarrior language reference which says, “A CHAR variable occupies two byrtes of storage, except in packed array and records.” In other words, Delphi defines the single, low-order byte, while CodeWarrior defines the char as two bytes. So, that high-order byte in the Delphi char is likely to not be zero, as you might expect.

To make sure I’m only looking at the character byte itself, I do this immediately upon entering the DLL routine:

```
procedure DoSomething(k:char; var
num:integer);
begin
  k := CHR(ORD(k) mod 256);   { consider
only the lower byte}
  num := ORD(k);
end;
```

## String Types

Passing strings between the host program and the DLL is another challenge. Both Delphi and Visual Basic are known to handle certain data types in very odd ways, and I suspect that strings fall into this category.

First of all, we must note that Delphi has a few different kinds of string types. The “standard” Pascal string that we Mac types are used to is defined by Delphi under the name “ShortString”. Delphi also defines a “long string” which is a zero-terminated C-style string. Just to trap the unwary, Delphi usually defines the type string as a long string. (There is a $H compiler switch to change the meaning of this.)

To keep my DLL source code as catholic as possible, I’ve been declaring my own “ShortString” type on the Mac side, which helps to remind me of the Delphi quirks in this.

The scary part of the story is that Delphi and CodeWarrior seem to have different ways of treating strings when it comes to passing them as parameters. For example, to pass a string as a parameter, we can define the string as a var on the DLL side, but we do not do that on the host Delphi side. Delphi always passes a string parameter as an implicit pointer, and that appears to match what CodeWarrior does when we use var. However, if we declare the string var on both sides, the host app will crash. This certainly warrants further study. But for now, here is what works:

In CodeWarrior, we declare a routine like this:

```
type
  ShortString = string[255];
procedure ProcessThisText (var
sss:ShortString); DLLEXPORT;
```

In Delphi, we declare this call as:

```
procedure ProcessThisText(sss:
ShortString); pascal; external ‘CALCS.DLL’;
```

We can then make use of this call with code like this:

```
sss := 'Sample Text';
ProcessThisText(sss);
```

If you get nervous about the lack of symmetry by having var only on one side, there is another solution: we can pass a pointer to the string. Here’s how: In the CodeWarrior code for the DLL, we define a pointer as the type for this parameter.

```
type
  ShortString = string[255];
  StrPtr = ^ShortString;
procedure ProcessThisText(p:StrPtr);
DLLEXPORT;
```

Note that here on the DLL side, I'm giving an explicit pointer type for a string, so that we can easily de-reference it. As an example, the following segment simply changes the second character of the received string to an "X":

```
procedure ProcessThisText (p:StrPtr);
begin
  p^[2] := 'X';
end;
```

Now let's move over to the Delphi host app. This time, I declare the DLL calls using just plain pointers. For example:

```
procedure ProcessThisText(p:pointer);
pascal; external 'CALCS.DLL';
```

Using a plain pointer makes it easy to use this call in the Delphi code, without having to coerce types constantly. Here is an example:

```
var
  sss: ShortString;
begin
  sss := 'Sample Text';
  ProcessThisText(@sss);
end;
```

Note that since we are passing a pointer, we can anticipate that the called routine in the DLL may modify the string, and we'll see that modification when the call returns back in the main program.

### Boolean Arrays

In CodeWarrior, a Boolean array is stored one Boolean bit per bytes, and a packed array of Boolean is stored as a bitstring. A packed array of 16 Booleans would require two bytes for storage. Delphi is different. Even in a packed array, a Boolean requires one byte, so a packed array of 16 Booleans requires 16 bytes. Conclusion: You can't pass a packed Boolean array in this way. Either use array of Boolean (not packed) or use bit operators to get at the bit you want.

Also, be aware that Pascal in CodeWarrior Pro version 1 has serious bugs involving packed arrays of booleans. You should use CW Pro 2 or later instead.

## Calling Conventions

You thought it was nasty accounting for the differences between calling conventions for C and Pascal? Well, the rules for DLL make that sound trivial. There are variations to establish the order in which parameters are passed, which piece of code does cleanup, whether names are "decorated", how the case to be used for names is determined, and whether registers are used.

If you want to cut to the chase, I'll give you the "answers" right away: When creating a DLL in CodeWarrior, use the default calling convention (that is, do nothing special). When using that DLL from Delphi, be sure to use the keyword pascal in all declarations. Now, if you want the details about this match-up, read on!

The best sources for information calling conventions as used by CodeWarrior and Delphi are these:

- "Inside CodeWarrior Professional: Pascal Compiler Guide", in the section, "Calling Conventions for Windows95/NT"
- Borland Delphi 3, "Object Pascal Language Guide", chapter 19, "Control Issues"

The DLLs that comprise Windows itself use a convention that is defined by Microsoft and is named stdcall. As a result, this convention would seem to be the closest thing to a lowest common denominator. Microsoft also defines two other conventions, cdecl and fastcall.

Delphi offers five different directives: register, pascal, cdecl, stdcall, and safecall. Interestingly, the default is register. Note what this means! If you do not specify calling convention, you will generate code that uses a different convention than that of Windows.

Metrowerks CodeWarrior supports stdcall, cdecl, fastcall and its own unnamed convention. The default is that unnamed convention. Once again, if you do not address the issue of calling convention, your CodeWarrior DLL is likely to run into problems if you try to use it in Delphi!

Delphi's register convention and Microsoft's fastcall convention (which is supported by CodeWarrior) both attempt to pass some parameters via registers, and this conceptually can improve performance. However, the two conventions appear to be defined differently, so they cannot be used together to pair a DLL with an app that uses it.

This table summarizes and contrasts the calling conventions defined or supported by Windows, Metrowerks and Delphi:

| Defined by | Name | BD | CW | Argument Order | Uses Registers | Stack Clean-up | Name Decoration | Case Translation |
|---|---|---|---|---|---|---|---|---|
| Microsoft | cdecl | √ | √ | Right-to-left | No | Caller | _name | none |
| Microsoft | stdcall | √ | √ | Right-to-left | No | Routine | _name@391 | none |
| Microsoft | fastcall | | √ | Right-to-left | Yes2 | Routine | @name | none |
| CodeWarrior | (none) | | √ √ | Left-to-right | No | Routine | name | 1st appearance |
| Delphi | pascal | √ | | Left-to-right | No | Routine | ? | ? |
| Delphi | register | √ √ | | Left-to-right | Yes3 | Routine | ? | ? |
| Delphi | safecall | √ | | Right-to-left | No | Routine | ? | ? |

***BD*** *– Supported in Borland Delphi*

***CW*** *– Supported in Metrowerks CodeWarrior*

√ *– Environment supports this convention. The default convention is shown with an outline-style checkmark.*

√ √ *– Default. Note that CodeWarrior and Delphi have different defaults, and neither is a "standard" defined by Microsoft. Therefore, if you don't take calling conventions into account, your chance of getting these guys to talk successfully to each other is near zero.*

*1 See discussion below.*

*2 The fastcall convention uses registers for the first two arguments that are four bytes or smaller in size.*

*3 Delphi 's register convention passes three arguments in registers.*

If you intend to pass arguments between routines in a Delphi host and a CodeWarrior DLL, it is imperative that you define the same calling convention on both sides. The obvious choice to use is stdcall: it is defined by Microsoft, it is the scheme used by Windows itself, and it is supported by both Delphi and CodeWarrior.

However, there is a problem with stdcall: name decoration. If you create a DLL with CodeWarrior using the stdcall convention, the actual routine names in the DLL will be "decorated" as shown in the table above. For example, suppose you write a routine named "AddNumbers" which has 20 bytes worth of parameters; the actual name carried within the DLL would be "_AddNumbers@20". The problem is that Borland Delphi does not recognize these decorated names. When you try to launch the Windows app, Delphi complains that the routine is missing, with a message like this:

```
Error Starting Program:
The MYPROJ.EXE file is
linked to missing export
MYDLL.DLL:AddNumbers.
```

If you try to access a routine explicitly named "_AddNumbers@20" then that will work, but that's clearly going to break the next time you revise the calling sequence.

(Windows95 supplies a command "QuickView" that can tell you details about what lurks inside of a DLL. This is how to determine what the actual routine names are within the DLL. To use QuickView, point to the DLL file icon, and right-click.)

It's a trifle suspicious that only the CodeWarrior manual seems to discuss the issue of name decoration. The DLLs that comprise Windows itself are supposedly written to stdcall, but the routines inside do not have name decoration, despite the CodeWarrior's implication that name decoration is a basic part of stdcall.

In inspecting the above table, there seems to be only one other match between the calling conventions of CodeWarrior and Delphi. This is the "pascal" convention of Delphi, which seems to match the default convention of CodeWarrior. This is indeed the solution! We take the default in CodeWarrior, and then apply the pascal convention in Delphi.

## Shaking Hands

Here is how I declare my functions and procedures in the CodeWarrior code that creates the DLL:

```
function AddSingle(i,j:real): real;
{ $IFC DLLTARGET}
  DLLEXPORT;
{ $ENDC}
```

Note that we take the default calling convention by not using any explicit convention keyword. When we are building a DLL, the DLLTARGET compiler variable will be true, and the DLLEXPORT keyword will be applied. If for some other purpose you do need to use an explicit calling convention, the keyword goes within the IFC test, just before the DLLEXPORT keyword.

After we take the DLL over to Windows, we can access it in Borland Delphi with a statement like this:

```
function AddSingle(a,b:single): single;
pascal; external 'CALCS.DLL';
```

The "pascal" keyword matches the default calling convention used by CodeWarrior. Note that I am careful to use the same 4-byte floats on both sides, "real" on CodeWarrior and "single" on Delphi.

## Using Another Convention

If you intend to use some convention other than the CodeWarrior default, you should know that the Inside CodeWarrior Professional: Pascal Compiler Guide gives incorrect information on how to invoke these conventions. To export a routine using the default calling convention, the statement looks like this:

```
function AddSingle(i,j:real): real;
DLLEXPORT;
```

To use the stdcall convention instead, the manual talks about ":_stdcall", but that seems to have been copied over incorrectly from the C documentation. The correct way to do this in Pascal is with a simple keyword, like this:

```
function AddSingle(i,j:real): real;
stdcall; DLLEXPORT;
```

I assume the other conventions are invoked in a corresponding way.

## Files Produced by CodeWarrior

When constructing a DLL, CodeWarrior emits two files. If you are create a DLL named "CALCS", say, you will get the DLL itself, "CALCS.DLL" and also a file with the name "CALCS.DLL.lib". The purpose of the second file is confusing, and even some of Metrowerks' support people got sidetracked by this and gave erroneous advice.

That "lib" file is used when building a second CodeWarrior project that makes use of the DLL. The key fact about these files is this: the "filename.DLL.lib" is usable only by CodeWarrior, and is both unnecessary and irrelevant for use of the DLL from anywhere else. Borland Delphi simply investigates the DLL itself at compile-time to pick up the same information.

So, to build DLLs in CodeWarrior for use in Delphi, just ignore the “xxx.DLL.lib” file.

## Mac Pascals vs. Delphi Pascal

While the crossplatform library approach works well, you’ll need to program the interface of your Windows app in Delphi directly. At that point, you must deal with differences between Delphi and other Pascals.

Knowing Pascal on the Mac, especially Object Pascal, will be vastly useful to you in learning Delphi. You’ll be writing juicy Delphi code quickly and you’ll also discover that large chunks of your non-interface Mac code can be ported in toto. However, Delphi’s flavor of Pascal is not the same as Think / CodeWarrior / MPW. There are syntactic differences that must be addressed. In fact, according to David Intersimone, one of the key Delphi designers, the reason Pascal was chosen as the basis for Delphi was because “we [Borland] own it” and could modify the language with minimal complaints. A new Delphi programmer coming from Mac would do well to review the language manual carefully. Just a few examples:

Delphi allows two units to cross-reference each other, as long as at least one has the “uses” clause in its implementation section. Two object definitions can refer to each other, but only if they are within the same “type” section.

A continuing annoyance in moving Pascal code to Delphi is that it doesn’t support StringOf, Concat, “&” or “ | ”. On the other hand, Delphi does provide “+” to concatenate strings.

## Delphi/Windows Oddities

Mac Pascal programmers are spoiled by the elegance of Think Pascal and the power of CodeWarrior (too bad we don’t have that in one package). Although Delphi is amazing, there is a decided lack of the crisp polish we’re used to. There is a single source file window, with tabs horizontally to choose a file. I guarantee you will despise the Delphi editor; I do most of my serious Delphi coding using the CodeWarrior editor.

The Delphi IDE keybindings are not configurable, there are no real resources, the number of files explodes with little effort, only one project can be open at a time, and every “form” has its own source unit. Debugging and the object browser are pathetically inferior to Think Pascal’s.

Delphi comes with a very rich set of components, with a lively 3rd party market as well. But there is danger of a “close enough” syndrome: using an existing component that is suitable, but not ideal, for a given purpose. And since components are compiled separately, it’s not easy to examine source code to quickly modify a subclass.

## When In Rome ... Doing Things the Delphi Way

After using RAD, you may never want to go back. The allure of using RAD is so captivating that it softens the blow of having to be involved with Windows. To take advantage of the power, the Mac programmer must re-think some issues. Here are some examples:

Delphi provides hooks for “event handlers”, routines that are called when certain events occur. When used well, this streamlines and clarifies code. For example, rather than calling some needed routine when an interface action happens, attach code to the “OnChange” event. Just add the data to the list whereever needed, but they have your clean-up code in the OnChange handler. You’ll quickly get very addicted to having these event hooks available, and it’s frustrating when you expect one that turns out to not exist (such as when a cell is edited in a grid).

It takes time to warm to the notion of storing your data within the visual components themselves. For example, a checkbox itself is used to store a setting rather than being used to set a value somewhere else. Doing this makes it easy to rename or rearrange components.

Delphi's OnIdle is not the same as a Mac null event. When processing an OnIdle message, if you display any alerts then additional OnIdle triggers will occur as the user manipulates the alert.

A very major difference is that almost all actions of Delphi interface components wind up being handled by methods of the form rather than of the component involved. And, since each form has its own unit, this makes it awkward to organize methods by function rather than form. Coming from the Mac, this all seems very odd: it seems reasonable and proper that a button should handle its own clicks. And yet, if that click causes some chain of events to occur, it indeed would be better to have a bit of indirection, in which the button calls some method (of something else). That way, it would be easy to also call the same method if we wish to extend the app with a new command, scripting support or interapp messaging. The practice in Delphi is not wrong, it is just different and it may be better.

## Delphi and "Properties"

A very significant difference between Delphi and other Pascals is the idea of "properties" of an object class. Both properties and fields are used in Delphi, and to the first-time observer, they appear to be used in the same way. To wit, both might have assignments like this:

```
pondscum.density := 1.34;
pondscum.color := kPutridGreen;
```

Now here's the wacky part. A "field" is a real instance variable, but a "property" merely acts like one. The definition of a property (within the class definition) looks like this:

```
property color:TColor read GetColor write
SetColor;
```

When an assignment statements appears to be assigning the value of a property, what it's really doing is calling the method specified by the "write" part of the definition. (And a statement like "itsColor := pondscum.color;" is compiled into a call to the method defined by the "read" part.)

So, an assignment like this:

```
pondscum.color := kPutridGreen;
```

is actually executed as though it were this:

```
pondscum.SetColor(kPutridGreen);
```

The code for "SetColor" can do anything. For example, it could accept the change in the setting, but also trigger a "repaint" event (like an inval). Thus, when you assign a new value to color, not only does it change the internal setting, it also refreshes the screen to show the change!!!!

Properties are also shown on the "object inspector" at design-time, and so when you are working on an interface and revise a property, the dialog being designed instantly reflects changes through this mechanism.

With properties, simple assignments become very powerful, and they encourage you to protect variables and go through methods instead, as is good practice. However, properties do add yet another non-standard Borland quirk to the language, and it can be a challenge to revise a variable without triggering these other side-effects.

## Suggested Sources for Delphi Information

It's been said, "There are no amateur Mac programmers, since programming the Mac requires professional skills". That sounded like a macho boast until I started programming Delphi and went looking for technical answers. There are a lot of newbie programmers in Delphi, which says a lot about the accessibility and power of the environment but makes it tough to find solid sources of nitty-gritty info. Here are some sources I've found valuable:

- Delphi tech info website:
  http://www.inprise.com/devsupport/delphi/ti_list/

- UNDU website:
  http://www.undu.com/

- 3rd party websites: info, free components
  http://www.amano-blick.com/~gnunn/gexperts.htm
  http://members.tripod.com/~delphipower/
  http://www.delphideli.com/quickmap.htm
  http://www.cyberramp.net/~jayres/
  http://www.cswnet.com/~choate/dex/
  http://www.westend.de/~hoerstemeier/
  http://www.delphi-jedi.org/

- "Tomes of Delphi": While Delphi provides a rich set of libraries for all manner of system functions, at some point you're going to want to call native Windows functions. This is a problem for Pascal programmers, because virtually all books on Windows programming use examples in C (or its descendents). But now there are two new books that document the Windows calls in Pascal. The "Tomes of Delphi" are in two volumes (Wordware Publishing); one covers graphics, and the other covers other core libraries. Although the authors focus on calling Windows from Delphi, in truth there is very little Delphi-specific information in the books and they can be used just as effectively by CodeWarrior programmers.

- Easily overlooked, the CodeWarrior distribution includes a goodly number of Pascal examples for Windows. You'll find them in CodeWarrior Examples:Win32_x86 Examples:Pascal.

## Conclusion

Not only is it possible to write a single Pascal unit that can be used in both a CodeWarrior and Delphi project, it's practical and convenient as well. On the Windows side, an experienced Mac Pascal programmer tackling Delphi for the first time will find much of it to be familiar territory, and the appeal and power of that environment more than compensates for its quirks.