Writing Portable Code

CHAPTER

4

IN THIS CHAPTER

If you're upgrading to Delphi 6 from a previous version or want to maintain compatibility among Delphi versions, this chapter is written for you. The first section of this chapter discusses general compatibility issues you will face in moving between any versions of Delphi. In the second section, you'll find hints and tips for maintaining compatibility between Delphi on the Win32 platform and Kylix on the Linux platform. The remainder of the chapter highlights the often subtle differences between the various versions and how to take these differences into account in writing portable code or migrating between versions. Although Borland makes a concerted effort to ensure that your code is compatible between versions, it's understandable that some changes have to be made in the name of progress, and certain situations require code changes if applications are to compile and run properly under the latest version of Delphi.

General Compatibility

A number of issues affect general compatibility between the various versions of Delphi, C++Builder, and Kylix. By making yourself aware of the support built into the compiler for writing compatible code, as well as some of the common gotchas, you'll be well on your way to targeting multiple versions from a single code base.

Which Version?

Although most Delphi code will compile for all versions of the compiler, in some instances language or VCL differences require that you write slightly differently to accomplish a given task for each product version. Occasionally, you might need to be able to compile for multiple versions of Delphi from one code base. For this purpose, each version of the Delphi compiler contains a `VER`*`xxx`* conditional define for which you can test in your source code. Because Borland C++Builder and Kylix also ships with new versions of the compiler, these edition also contain this conditional define. Table 4.1 shows the conditional defines for the various versions of the Delphi compiler.

TABLE 4.1 Conditional Defines for Compiler Versions

Product	*Conditional Define*
Delphi 1	`VER80`
Delphi 2	`VER90`
C++Builder 1	`VER95`
Delphi 3	`VER100`
C++Builder 3	`VER110`
Delphi 4	`VER120`
C++Builder 4	`VER120`

TABLE 4.1 Continued

Product	*Conditional Define*
Delphi 5	VER130
C++Builder 5	VER130
Kylix 1	VER140
Delphi 6	VER140

Using these defines, the source code you must write in order to compile for different compiler versions would look something similar to this:

```
{$IFDEF VER80}
  Delphi 1 code goes here
{$ENDIF}
{$IFDEF VER90}
  Delphi 2 code goes here
{$ENDIF}
{$IFDEF VER95}
  C++Builder 1 code goes here
{$ENDIF}
{$IFDEF VER100}
  Delphi 3 code goes here
{$ENDIF}
{$IFDEF VER110}
  C++Builder 3 code goes here
{$ENDIF}
{$IFDEF VER120}
  Delphi 4 and C++Builder 4 code goes here
{$ENDIF}
{$IFDEF VER130}
  Delphi and C++Builder 5 code goes here
{$ENDIF}
{$IFDEF VER140}
  Delphi 6 and Kylix code goes here
{$ENDIF}
```

NOTE

If you're wondering why the Delphi 1.0 compiler is considered version 8, Delphi 2 version 9, and so on, it's because Delphi 1.0 is considered version 8 of Borland's Pascal compiler. The last Turbo Pascal version was 7.0, and Delphi is the evolution of that product line.

Units, Components, and Packages

The binary format of Delphi compiled units (`.dcu` files) tends to differ from compiler version to compiler version. This means that if you want to use the same unit in multiple versions of Delphi, you must have either binary units built for that specific compiler version or the source code to those units so that they can be recompiled. Bear in mind that if you use any custom components in your application—your own components or those developed by third parties—you must have the source to these components. If you don't have the version-specific binary or the source code to a particular third-party component, contact your vendor for a version of the component specific to your version of Delphi.

NOTE

This issue of compiler version versus unit file version isn't a new situation and is the same as C++ compiler object file versioning. If you distribute (or buy) components without source code, you must understand that what you're distributing or buying is a compiler-version–specific binary file that will probably need to be revised to keep up with subsequent compiler releases.

What's more, the issue of DCU versioning isn't necessarily a compiler-only issue. Even if the compiler weren't changed between versions, changes and enhancements to core VCL would probably still make it necessary that units be recompiled from source.

Delphi 3 introduced *packages*, the idea of multiple units stored in a single binary file. Starting with Delphi 3, the component library became a collection of packages rather than one massive component library DLL. Like units, packages aren't compatible across product versions, so you'll need to rebuild your packages for each version of Delphi, and you'll need to contact the vendors of your third-party components for version-specific packages.

IDE Issues

Problems with the IDE are likely the first you'll encounter as you migrate your applications. Here are a few of the issues you might encounter on the way:

- Delphi debugger symbol files (RSM) are not always compatible across versions. You'll know you're having this problem when you see the message `"Error reading symbol file."`. If this happens, the fix is simple: Rebuild the application.
- Starting with version 5, Delphi defaults to storing form files in text mode. If you need to maintain DFM compatibility with earlier versions of Delphi, you'll need to save the forms files in binary instead. You can do this by unchecking New Forms As Text on the Preferences page of the Environment Options dialog box.

- Code generation when importing and generating type libraries often changes from version to version. As of Delphi 5, you can customize type-library–to–Pascal symbol name mapping by editing the `tlibimp.sym` file. For directions, see the "Mapping Symbol Names in the Type Library" topic in the online help.

Delphi-Kylix Compatibility

If you endeavor to build applications with any degree of portability between Delphi and Kylix, the most important thing to realize is that VCL is a Windows-specific technology. If you want to build cross platform applications and components, you should use the Component Library for X-platform (CLX), which is currently supported until Delphi 6 and Kylix. CLX is described in greater detail in Chapters 10, "Component Architecture: VCL and CLX," and 13, "CLX Component Development." CLX can be broken down into four major components:

- BaseCLX, which contains the core portions of the component framework.
- DataCLX, which employs the dbExpress technology to provide efficient, lightweight data access and management. dbExpress is described in detail in Chapter 8, "Database Development with dbExpress."
- NetCLX, which provides components and wizards for creating network clients and servers. Perhaps most notably, NetCLX provides a very robust Web development application framework that encompasses and includes the WebBroker technology from previous versions. NetCLX allows targeting of Linux or Windows clients and servers.
- VisualCLX, which provides the cross-platform GUI capability. VisualCLX is externally very similar to VCL, but internally uses Troll Tech's (`http://www.trolltech.com`) Qt library (as opposed to the Win32 API like in VCL). Qt is a cross-platform GUI framework that enables developers to target a variety of platforms, including Windows and Linux.

When you create a new CLX application using File, New, CLX Application and view the uses clause of the resulting main form unit, you will see a number of unit names beginning with the letter Q, such as `QGraphics`, `QControls`, `QForms`, and so on. These units are similar in content and function to the similarly named VCL units, although they are cross platform.

NOTE

Although the current versions of CLX support only Windows and Kylix, it is designed such that it can be extended relatively easily to other platforms. Qt, for example, supports about a dozen different platforms.

Not in Linux

Of course, you won't find the Windows-specific technologies you might have grown to know and love on the Linux platform. This means that technologies such as ADO, COM/COM+, BDE, and MAPI (among others) have no place in a cross-platform application. You should therefore avoid using units such as `Windows`, `ComObj`, `ComServ`, `ActiveX`, and `AdoDb` and platform-specific functions such as any WIn32 API call, `RaiseLastWin32Error()`, `Win32Check()`, and so on. Additionally, there are a number of technologies found in Delphi 6 that aren't available in Kylix 1 but will likely be found in future versions of Kylix. These include DataSnap, BizSnap (SOAP), and WebSnap technologies.

Compiler/Language Features

Although the Delphi and Kylix compilers both target the x86 processor architecture, there are a number of key differences in the compiler that you should be aware of in building portable applications.

`LINUX` Define

The Kylix compiler defines the `LINUX` conditional, whereas Delphi defines `MSWINDOWS` and `WIN32`, so that you can `IFDEF` your code in order to maintain platform-specific code in a single unit. Such code would like something like this:

```
{$IFDEF LINUX}
  // Linux-specific code goes here
{$ENDIF}
{$IFDEF MSWINDOWS}
  // Windows-specific code goes here
{$ENDIF}
```

PIC Format

The Linux compiler produces executables in Position Independent Code (PIC) format, which is a slight variation on the type of code produced by the Windows compiler. Although this change has little or no effect if you're just writing Pascal code, it can have a dramatic impact on externally linked assembler modules or built-in assembler. Most notably, PIC requires access to all global data to be relative to the EBX register, so the following line in Delphi

```
mov eax, SomeVar
```

would be written for PIC as

```
mov eax [ebx].SomeVar
```

Because of the heavy reliance on the EBX register, PIC also requires that the value of EBX be preserved across function calls and restored prior to external calls. If you want to `IFDEF` your

built-in assembly code for PIC and non-PIC, the compiler also defines a PIC conditional for which you can check:

```
{$IFDEF PIC}
  // PIC specific code goes here
{$ENDIF}
```

Calling Conventions

It's worth noting that `stdcall` and `safecall` calling conventions don't exist in Kylix. These directives simply map to the `cdecl` calling convention in Kylix. This is generally only an issue if you have assembly code that depends on parameter order and stack cleanup.

Platform-isms

In general, you should be wary of hard-coding platform-isms, or platform-specifics idioms into your applications. Some items in the vein to keep in mind include

- The notion of drive letters does not exist on Linux.
- The directory separator is a backslash (`\`) on Windows and a forward slash (`/`) on Linux. Delphi's `PathSeparator` constant will show you which to use.
- The directory list delimiter is a semicolon (`;`) on Windows and a colon (`:`) on Linux.
- UNC pathnames exist only on Windows.
- Avoid depending on platform-specific directories, such as `c:\winnt\system32` or `/usr/bin`.

New Delphi 6 Features

A number of nice additions to Delphi 6, particularly in the language and compiler area, can make application development go more smoothly. However, it's important to bear in mind that employing these features might mean that your code will not compile in earlier product versions.

Variants

Rather than being implemented within the compiler, support for the `Variant` data type has been opened up to support user-installable types. This support is found in the `Variants` unit.

Enum Values

In an effort to achieve greater compatibility with C++, the compiler now supports the assignment of values to elements of an enumerated type, as shown here:

```
type
  TFoo = (fTwo=2, fFour=4, fSix=6, fEight=8);
```

$IF Directive

One particular feature that is a long time coming is the addition of the $IF and $ELSEIF directives that allow you to check for defined symbols and to perform Boolean comparisons against constants, as shown here:

```
{$IF Defined(MSWINDOWS) and SomeConstant >= 6}
  // do something
{$ELSEIF SomeConstant < 2}
  // do something else
{$ELSE}
  // if all else fails
{$ENDIF}
```

Potential Binary DFM Incompatibility

The mechanism that saves and loads Delphi forms from stream has been modified, particularly as it relates to high ASCII characters (those higher than 127). Binary DFMs containing high ASCII characters might not be readable in earlier Delphi versions. A workaround would be to use the text version of the form.

Migrating from Delphi 5

Although compatibility between Delphi 5 and 6 is quite good, there are a few minor issues you should be aware of as you make the move.

Writable Typed Constants

The default state of the $J compiler switch (also known as $WRITEABLECONST) is now off, where it was on in previous versions. This means that attempts to assign to typed constants will raise a compiler error unless you explicitly enable this behavior using $J+.

Cardinal Unary Negation

Prior to Delphi 6, Delphi used 32-bit arithmetic to handle unary negation of Cardinal type numbers. This could lead to unexpected results. Consider the following bit of code:

```
var
   c: Cardinal;
   i: Int64;
begin
   c := 4294967294;
   i := -c;
   WriteLn(i);
end;
```

In Delphi 5, the value of `i` displayed would be 2. Although this behavior is incorrect, you might have code that relies on this behavior. If so, you should know that Delphi 6 has corrected this issue by promoting the Cardinal to an Int64 prior to performing the negation. The final value of `i` displayed in Delphi 6 is `4294967294`.

Migrating from Delphi 4

This section highlights some of the issues you can expect if you're moving up from Delphi 4.

RTL Issues

The only issue you're likely to come across here deals with the setting of the floating-point unit (FPU) control word in DLLs. Prior to version 5, DLLs would set the FPU control word, thereby changing the setting established by the host application. Now, DLL startup code no longer sets the FPU control word. If you need to set the control word to ensure some specific behavior by the FPU, you can do it manually using the `Set8087CW()` function in the `System` unit.

VCL Issues

There are a number of VCL issues that you may come across, but most involve some simple edits as a means to get your project on track. Here's a list of these issues:

- The type of properties that represent an index into an image list has changed from `Integer` to `TImageIndex` type between Delphi 4 and 5. `TImageIndex` is a strongly typed `Integer` defined in the `ImgList` unit as

  ```
  TImageIndex = type Integer;
  ```

  This should only cause problems in cases where exact type matching matters, such as when you're passing `var` parameters.
- `TCustomTreeview.CustomDrawItem()` added a `var` parameter called `PaintImages` of type `Boolean`. If your application overrides this method, you'll need to add this parameter in order for it to compile in Delphi 5 or higher.
- If you're invoking pop-up menus in response to `WM_RBUTTONUP` messages or `OnMouseUp` events, you might exhibit "double" pop-up menus or no pop-up menus at all when compiling with Delphi 5 or later. Delphi now uses the `WM_CONTEXT` menu message to invoke pop-up menus.

Internet Development Issues

If you're developing applications with Internet support, we have some bad news and some good news:

- The `TWebBrowser` component, which encapsulates the Microsoft Internet Explorer ActiveX control, has replaced the `THTML` component from `Netmasters`. Although the `TWebBrowser` control is much more feature rich, you're faced with a good deal of rewrite if you used `THTML` because the interface is totally different. If you don't want to rewrite your code, you can go back to the old control by importing the `HTML.OCX` file from the `\Info\Extras\NetManage` directory on the Delphi CD-ROM.
- Packages are now supported when building ISAPI and NSAPI DLLs. You can take advantage of this new support by replacing HTTPApp in your `uses` clause with WebBroker.

Database Issues

A few database issues might trip you up as you migrate from Delphi 4. These involve some renaming of existing symbols and the new DataSnap architecture (formerly called MIDAS):

- The type of the `TDatabase.OnLogin` event has been renamed `TDatabaseLoginEvent` from `TLoginEvent`. This is unlikely to cause problems, but you might run into troubles if you're creating and assigning to `OnLogin` in code.
- The global `FMTBCDToCurr()` and `CurrToFMTBCD()` routines have been replaced by the new `BCDToCurr` and `CurrToBCD` routines (and the corresponding protected methods on `TDataSet` have been replaced by the protected and undocumented `DataConvert` method).
- DataSnap (formerly MIDAS) has undergone some significant changes since Delphi 4. See Chapter 21, "DataSnap Development," for information on the changes and new features.

Migrating from Delphi 3

Although there aren't a great deal of compatibility issues between Delphi 3 and later versions, the few issues that do exist can be potentially more problematic than porting from any other previous version of Delphi to the next. Most of these issues revolve around new types and the changing behavior of certain existing types.

Unsigned 32-bit Integers

Delphi 4 introduced the `LongWord` type, which is an unsigned 32-bit integer. In previous versions of Delphi, the largest integer type was a signed 32-bit integer. Because of this, many of the types that you would expect to be unsigned, such as `DWORD`, `UINT`, `HResult`, `HWND`, `HINSTANCE`, and other handle types, were defined simply as `Integers`. In Delphi 4 and later, these types are redefined as `LongWords`. Additionally, the `Cardinal` type, which was previously a subrange type of `0..MaxInt`, is now also a `LongWord`. Although all this `LongWord` business won't cause problems in most circumstances, there are several problematic cases you should know about:

- `Integer` and `LongWord` are not `var`-parameter compatible. Therefore, you cannot pass a `LongWord` in a `var Integer` parameter, and vice versa. The compiler will give you an error in this case, so you'll need to change the parameter or variable type or typecast to get around this problem.
- Literal constants having the value of `$80000000` through `$FFFFFFFF` are considered `LongWord`s. You must typecast such a literal to an `Integer` if you want to assign it to an `Integer` type. Here's an example:

```
var
  I: Integer;
begin
  I := Integer($FFFFFFFF);
```

- Similarly, any literal having a negative value is out of range for a `LongWord`, and you'll need to typecast to assign a negative literal to a `LongWord`. Here's an example:

```
var
  L: LongWord;
begin
  L := LongWord(-1);
```

- If you mix signed and unsigned integers in arithmetic or comparison operations, the compiler will automatically promote each operand to `Int64` in order to perform the arithmetic or comparison. This can cause some very difficult-to-find bugs. Consider the following code:

```
var
  I: Integer;
  D: DWORD;
begin
  I := -1;
  D := $FFFFFFFF;
  if I = D then DoSomething;
```

  Under Delphi 3, *`DoSomething`* would execute because `-1` and `$FFFFFFFF` are the same value when contained in an `Integer`. However, because Delphi 4 and later will promote each operand to `Int64` in order to perform the most accurate comparison, the generated code ends up comparing `$FFFFFFFFFFFFFFFF` against `$00000000FFFFFFFF`, which is definitely not what's intended. In this case, *`DoSomething`* will not execute.

TIP

The compiler in Delphi 4 and later generates a number of new hints, warnings, and errors that deal with these types of compatibility problems and implicit type promotions. Make sure that you turn on hints and warnings when compiling in order to let the compiler help you write clean code.

64-Bit Integers

Delphi 4 also introduced a new type called `Int64`, which is a signed 64-bit integer. This new type is now used in the RTL and VCL where appropriate. For example, the `Trunc()` and `Round()` standard functions now return `Int64`, and there are new versions of `IntToStr()`, `IntToHex()`, and related functions that deal with `Int64`.

The Real Type

Starting with Delphi 4, the `Real` type became an alias for the `Double` type. In previous versions of Delphi and Turbo Pascal, `Real` was a six-byte, floating-point type. This shouldn't pose any problems for your code unless you have `Real`s written to some external storage (such as a `file of record`) with an earlier version or you have code that depends on the organization of the `Real` in memory. You can force `Real` to be the old 6-byte type by including the `{$REALCOMPATIBILITY ON}` directive in the units you want to use the old behavior. If all you need to do is force a limited number of instances of the `Real` type to use the old behavior, you can use the `Real48` type instead.

Migrating from Delphi 2

You'll find that a high degree of compatibility between Delphi 2 and the later versions means a smooth transition into a more up-to-date Delphi version. However, some changes have been made since Delphi 2, both in the language and in VCL, that you'll need to be aware of to migrate to the latest version and take full advantage of its power.

Changes to Boolean Types

The implementation of the Delphi 2 Boolean types (`Boolean`, `ByteBool`, `WordBool`, `LongBool`) dictated that `True` was ordinal value `1` and `False` ordinal value `0`. To provide better compatibility with the Win32 API, the implementations of `ByteBool`, `WordBool`, and `LongBool` have changed slightly; the ordinal value of `True` is now `-1` (`$FF`, `$FFFF`, and `$FFFFFFFF`, respectively). Note that no change was made to the `Boolean` type. These changes have the potential to cause problems in your code—but only if you depend on the ordinal values of these types. For example, consider the following declaration:

```
var
  A: array[LongBool] of Integer;
```

This code is quite harmless under Delphi 2; it declares an `array[False..True]` (or `[0..1]`) `of Integer`, for a total of three elements. Under Delphi 3 and later, however, this declaration can cause some very unexpected results. Because `True` is defined as `$FFFFFFFF` for a `LongBool`, the declaration boils down to `array[0..$FFFFFFFF] of Integer`, or an array of 4 billion `Integers`! To avoid this problem, use the `Boolean` type as the array index.

Ironically, this change was necessary because a disturbing number of ActiveX controls and control containers (such Visual Basic) test BOOLs by checking for -1 rather than testing for a zero or nonzero value.

TIP

To help ensure portability and to avoid bugs, never write code like this:

```
if BoolVar = True then ...
```

Instead, always test Boolean types like this:

```
if BoolVar then ...
```

ResourceString

If your application uses string resources, consider taking advantage of ResourceStrings as described in Chapter 2, "The Object Pascal Language." Although this won't improve the efficiency of your application in terms of size or speed, it will make language translation easier. ResourceStrings and the related topic of resource DLLs are required to be able to write applications displaying different language strings but have them all running on the same core VCL package.

RTL Changes

Several changes made to the runtime library (RTL) after Delphi 2 might cause problems as you migrate your applications. First, the meaning of the HInstance global variable has changed slightly: HInstance contains the instance handle of the current DLL, EXE, or package. Use the new MainInstance global variable when you want to obtain the instance handle of the main application.

The second significant change pertains to the IsLibrary global. In Delphi 2, you could check the value of IsLibrary to determine whether your code was executing within the context of a DLL or EXE. IsLibrary isn't package aware, however, so you can no longer depend on IsLibrary to be accurate, depending on whether it's called from an EXE, DLL, or a module within a package. Instead, you should use the ModuleIsLib global, which returns True when called within the context of a DLL or package. You can use this in combination with the ModuleIsPackage global to distinguish between a DLL and a package.

TCustomForm

The Delphi 3 VCL introduced a new class between TScrollingWinControl and TForm called TCustomForm. In itself, that shouldn't pose a problem for you in migrating your applications

from Delphi 2; however, if you have any code that manipulates instances of `TForm`, you might need to update it so that it manipulates `TCustomForms` instead of `TForms`. Some examples of these are calls to `GetParentForm()`, `ValidParentForm()`, and any usage of the `TDesigner` class.

CAUTION

The semantics for `GetParentForm()`, `ValidParentForm()`, and other VCL methods that return `Parent` pointers have changed slightly from Delphi 2. These routines can now return `nil`, even though your component has a parent window context in which to draw. For example, when your component is encapsulated as an ActiveX control, it might have a `ParentWindow`, but not a `Parent` control. This means that you must watch out for Delphi 2 code that does this:

```
with GetParentForm(xx) do ...
```

`GetParentForm()` can now return `nil` depending on how your component is being contained.

`GetChildren()`

Component writers, be aware that the declaration of `TComponent.GetChildren()` has changed to read as follows:

```
procedure GetChildren(Proc: TGetChildProc; Root: TComponent); dynamic;
```

The new `Root` parameter holds the component's root owner—that is, the component obtained by walking up the chain of the component's owners until `Owner` is `nil`.

Automation Servers

The code required for automation has changed significantly from Delphi 2. Chapter 15, "COM Development," describes the latest process of creating Automation servers in Delphi. Rather than describe the details of the differences here, suffice it to say that you should never mix the Delphi 2 style of creating Automation servers with the more recent style found in Delphi 3 and later.

In Delphi 2, automation is facilitated through the infrastructure provided in the `OleAuto` and `Ole2` units. These units are present in later releases of Delphi only for backward compatibility, and you shouldn't use them for new projects. Now the same functionality is provided in the `ComObj`, `ComServ`, and `ActiveX` units. You should never mix the former units with the latter in the same project.

Migrating from Delphi 1

If you're lucky enough to still be maintaining code that must be compiled and run under both 16 and 32-bit Windows, you have our condolences. There are numerous points of incompatibility between Delphi 1 and later versions, ranging from most of the basic data types to VCL to the Windows API. Because of the relatively small number of developers who continue to maintain and develop 16-bit applications, that information isn't in the text of this book, but you'll find it in Chapter 15 of the electronic copy of *Delphi 5 Developer's Guide* on the CD accompanying this book.

Summary

Armed with the information provided by this chapter, you should be able to migrate your projects smoothly from any previous version of Delphi to Delphi 6. Also, with a bit of work, you'll be able to maintain projects that work with multiple versions of Delphi.