Volume 14, Number 5—May 2010 (Special Issue)

www.bcbjournal.com
ISSN 1093-2097

EnCoded Communications Group

C++BUILDER Developer's Journal

A Monthly Publication Offering Tips & Techniques for C++Builder

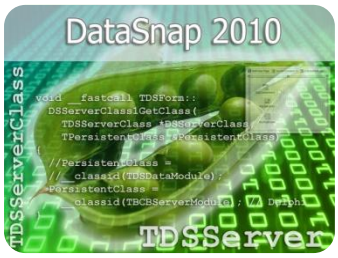
Special Issue on C++Builder 2010

Edited by Malcolm Smith, Malcolm Groves, Curtis Krauskopf, Remy Lebeau, Bob Swart, and Damon Chandler

New Archive CD 4.0 – Page 18

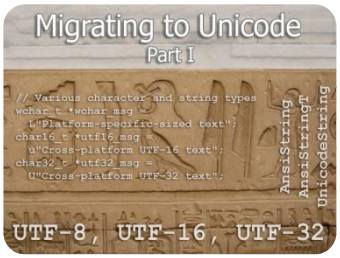
MJFAF Special Offer Page 28

In This Issue:



4

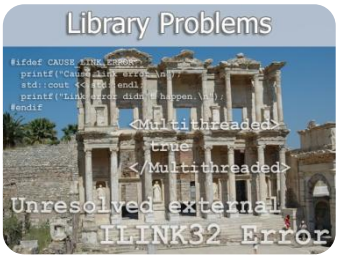
12



19

29

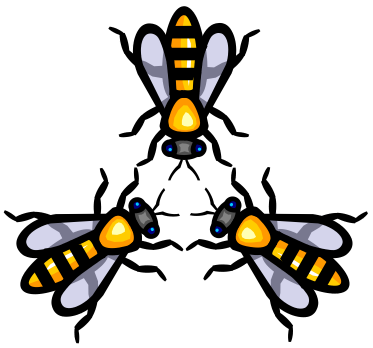
32



40

Special Issue Preface

By Malcolm Smith



Welcome to this special issue of the C++Builder Developer's Journal on C++Builder 2010.

C++Builder continues to be the tool of choice for developers seeking a powerful, yet easy-to-use application-development solution for Windows. Embarcadero C++Builder 2010 represents the latest and greatest release of this award-winning tool.

This special issue discusses several important aspects of C++Builder development which are unique to version 2010. Even more, many of the techniques presented in this issue are relevant to all developers who upgrade to a newer version of C++Builder.

This special issue discusses several important aspects of C++Builder development which are unique to version 2010.

Bob Swart opens the issue with an article on how to use C++Builder 2010 to build DataSnap 2010 server and client applications. Bob discusses several unique aspects of DataSnap 2010, including server methods. If you need a robust way of creating a data broker/client application, this is the article for you.

Next, **Josh Kelley** provides an important two-part series on migrating a C++Builder application to Unicode—a daunting process for most developers. In his first article, Josh provides an introduction to Unicode and discusses various ways to work with Unicode text in C, C++, the Windows API, and the VCL.

In his second article on Unicode, Josh provides specific details and guidelines on how to migrate a C++Builder application to Unicode. Josh presents several C/C++ techniques that can be used to help with a Unicode migration. This two-part series is a must-read for all C++Builder developers.

Next, **Curtis Krauskopf** provides three techniques for changing the default directory where new C++Builder 2010 projects are saved. As we all know, the C++Builder IDE insists on saving new projects into the folder "My Documents\RAD Studio\Projects," whereas most developers prefer to save their projects in custom locations. Curtis presents three approaches for customizing this location.

Next, **ByeongCheol Nam** and **Ki-Tae Bae** describe how to develop a multi-touch system using C++Builder 2010. The authors present a system which uses an infrared sensor as a multi-touch sensing device; however, for those without such a device, you can still test the code by using two mice.

Finally, **Curtis Krauskopf** closes the special issue with an article on his experiences in porting libraries from BCB 6 to C++Builder 2010. Curtis discusses how to overcome a significant potential problem when linking static libraries into a C++Builder 2010 project.

Thanks to the authors and editors who have made this special issue possible. A special thanks is particularly due to our loyal readers who have continued to support the C++Builder Developer's Journal since its first publication over 13 years ago.

On behalf of the special issue editorial board, I'd like to welcome you to this special issue, and I sincerely hope you find the material useful. Enjoy!

Malcolm Smith, Special Issue Lead Editor
MJ Freelancing and Comvision Pty Ltd

Other Special Issue Editors:

Guest Editor: **Malcolm Groves**
Embarcadero Technologies

Guest Editor: **Curtis Krauskopf**
The Database Managers

Guest Editor: **Remy Lebeau**
Lebeau Software, TeamB, and Indy Project Team

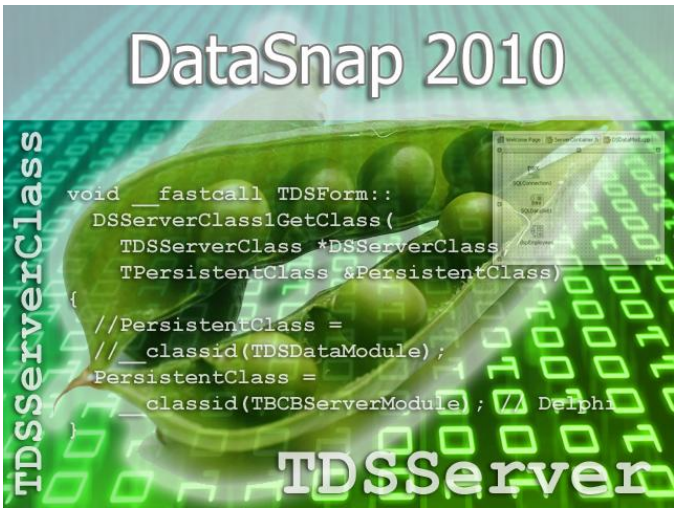
Guest Editor: **Bob Swart**
Bob Swart Training & Consultancy (eBob42.com)

Publications Editor: **Damon Chandler**
C++Builder Developer's Journal

DataSnap 2010

By Bob Swart

Versions: C++Builder 2010



In a previous article, I demonstrated how to use DataSnap in C++Builder 2007 on Windows Vista to build multi-tier database applications [1]. In the years that followed, DataSnap has undergone a significant overhaul. No longer based on COM, the new DataSnap multi-tier architecture is more lightweight, but also feels like it is "not yet finished" in all places.

Unfortunately, C++Builder support is one of the areas which is a lacking compared to the Delphi support for the new DataSnap. Delphi 2010 includes two DataSnap Wizards to produce different kinds of projects, for example, with all components and properties already connected and ready to go. With C++Builder, we still have to do all that manually, and some things will take more effort than others.

In this article I will take you all the way, from start to deployment, and even to server methods which *out-of-the-box* are not possible in C++ for C++Builder, but with a little help from Delphi (or a C++Builder helper class) we can still get them as well. Anyway, more than enough to cover lots of time and pages, so let's get started.

C++Builder 2010 Enterprise

You will need the Enterprise (or greater) edition of C++Builder to be able to participate. Since there are no special DataSnap wizards in C++Builder, we need to start with a new VCL Forms Application (you can also start with a Windows Service Application, but the normal VCL Forms Application is the easiest one to demonstrate and debug). When saving the project, I save the form (UnitX) in the file ServerContainer.cpp. This will be the unit that holds the DataSnap server communication components. The project itself is saved in DS2010BCBServer-.cbproj in my case.

I've set the `Name` property of the main form to DSForm, and the `Caption` to "C++Builder DataSnap 2010 Server Container", so we know what's going on when we see this form running.

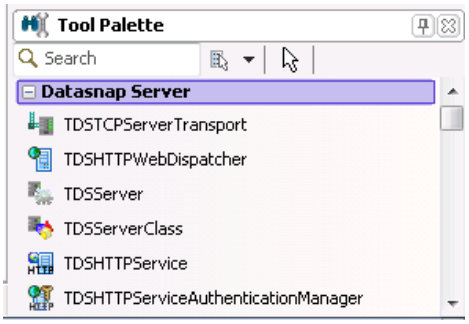


Figure 1: *DataSnap server components.*

The "DataSnap Server" category on the Tool Palette contains the new DataSnap components that we need to use to produce a DataSnap server application (see **Figure 1**). Two components are always needed for any and all DataSnap Server applications: the `TDSServer` and `TDSServerClass` components.

TDSServer

First, place a `TDSServer` component on the main form. The `TDSServer` is the actual "engine" of the DataSnap Server, and we can start, pause, or stop this engine using the corresponding commands. By default, the `TDSServer` has the `AutoStart` property set to true, to automatically start the engine when the application itself starts. There is also a property called `HideDSAdmin` which is set to false by default, but can be set to true to hide some of the methods that the DataSnap Server exposes that are not of use for the average DataSnap Server application. If you leave this property set to false, then you'll see the methods later when we connect the DataSnap client to the server.

The `TDSServer` class has five events that we can hook into, to trace what's going on with the server. The events are `OnConnect`, `OnDisconnect`, `OnError`, `OnPrepare` and `OnTrace`. The

OnTrace event gives us interesting details on the commands and data being sent back and forth between the clients and the server. Even if you do not want to trace the communications, you should at least write an event handler for the OnError event, to ensure that the server will be notified (or at least someone will be notified) of an error situation. The TDSErrorEventObject has a property, Error, with the exception that was causing the error, so the least we can do is present that information:

```
void __fastcall TDSForm::DSServer1Error(
  TDSErrorEventObject *DSErrorEventObject)
{
  ShowMessage(DSErrorEventObject->
    Error->Message);
}
```

Apart from the Error property, the DSErrorEventObject also has properties for the DbxContext, the Transport, the Server and the DbxConnection to inspect when needed to get more details on the error.

TDSServerClass

Now, place a TDSServerClass component next to it. Assign the Server property of the TDSServerClass to the TDSServer component.

The TDSServerClass is the component that tells the DataSnap Server which class type to "service", and how to service this type. The "how" is determined by the LifeCycle property, which by default is set to Session. The alternative values are Invocation and Server. This LifeCycle property defines how long an instance of the designated class will "live."

With LifeCycle set to Server, we will get a single instance of the server class used to handle all incoming requests. This means that the class should be thread safe by not using class data members, but only local variables inside methods to avoid threading issues. We can use an object like this to count the number of incoming requests (hits) for example.

The other extreme is the LifeCycle value of Invocation, which means that each incoming request will result in a new instance of the DataSnap server class, which handles the request and then gets killed again. This is truly stateless, since the server class will never know anything from a previous request.

The default setting of LifeCycle to Session means that all incoming requests from a single source (a Session) will get a single instance of the DataSnap server class. This instance will handle all requests from that source, so it's possible to maintain state (i.e. to ask for "the next 10 records", since the server will remember the previous position). The default setting of Session results in the easiest way to build DataSnap servers and clients, but not in the most scalable architecture. A few thousand concurrent requests in a time span of only one minute will be tough to handle by any DataSnap server if LifeCycle is set to Session (but a lesser problem for LifeCycle set to Server or Invocation).

We must use the OnGetClass event hander of the TDSServerClass to specify which class to use as the DataSnap server class. This should be a data module if we want to expose TDataSetProvider components—which usually is the case. So, let's add a data module first, and then get back to the form and implement the OnGetClass event handler.

DataSnap data module

To add a data module, do "File | New | Other" (or right-click on the DS2010BCBServer project and select "Add New | Other"), and in the Object Repository go to the C++Builder Files category and pick the Data Module. I've set the name of the new data module to DSDataModule, and saved it in the file DSDataMod.cpp. In the Project Options panel, open the Forms panel and remove DSDataModule from the Auto-create Forms list. The reason for this is that an instance will be created dynamically at runtime, based on the settings of the TDSServerClass' LifeCycle value, remember?

We can now add our data access components on the data module, exposing them using TDataSetProvider components. For our example, place a TSQLConnection, TSQLDataSet and TDataSetProvider on the data module. We can use the TSQLConnection component to connect to a database using dbExpress, for example to a SQL Server database (I leave it up to the reader to connect to a database here) like the Northwind example database.

Make sure to connect the SQLConnection property of the TSQLDataSet to the TSQLConnection component. Then, we can set the CommandType to ctQuery, ctTable or ctStoredProc and specify a SQL command, a tablename or a stored procedure name in the CommandText property. Using the Northwind database, I've used a ctQuery with CommandText as follows:

```
select  "EmployeeID", "FirstName",
  "LastName", "City", "Country"
from "Employees"
```

The TDataSetProvider has a DataSet property that needs to point to the dataset that we want to export to the clients, in this case SQLDataSet1. The name of the TDataSetProvider is important, since this is a name that the developers at the client side will see. And DataSetProvider1 doesn't mean a lot, so we should change that to a more sensible name like dspEmployees. This results in the data module at design-time shown in **Figure 2**.

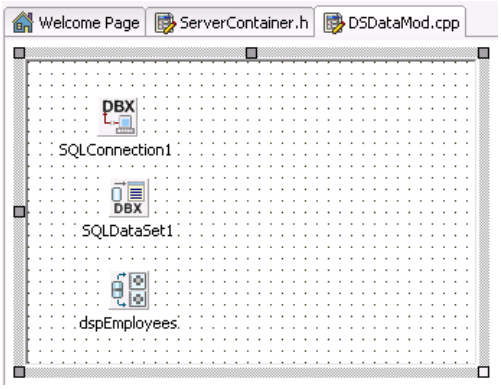


Figure 2: *DSDataMod.cpp.*

DataSnap server module

So far so good. Unfortunately, this is still only a data module, and for DataSnap 2010 we actually need something else: a TDSServerModule. The "New Server Module" wizard, however, is also not available for C++Builder developers, which is why we had to start with a TDataModule instead.

In order to transform the TDataModule into the required type, we need to make a few changes by hand in DSDataMod.h. First of all, add a line with

```
#include <DSServer.hpp>
```

Then, change the ancestor class of TDSDataModule from TDataModule to TDSServerModule:

```
class TDSDataModule : public
  TDSServerModule // was: TDataModule
```

Then, switch to the DSDataMod.cpp file, and make one modification to the source code: in the constructor, call TDSServerModule(Owner) instead of TDataModule(Owner), as follows:

```
__fastcall TDSDataModule::TDSDataModule(
  TComponent* Owner) :
  TDSServerModule(Owner)
  // was: TDataModule(Owner)
```

This will ensure that our TDSDataModule class is derived from TDSServerModule and not from TDataModule. (Note that you will be prompted by the IDE that it wants to correct the definition of the DFM file; you should re-save.)

OnGetClass

Now we can go back to the server container (main) form in ServerContainer.cpp. We need to make sure DSDataMod is used, to press Alt+F11 and add DSDataMod.cpp to the Header of the ServerContainer unit. We can now implement the OnGetClass event of the TDSServerClass component as follows:

```
void __fastcall TDSForm::
  DSServerClass1GetClass(
    TDSServerClass *DSServerClass,
    TPersistentClass &PersistentClass)
{
  PersistentClass =
    __classid(TDSDataModule);
}
```

This will make sure that the TDSDataModule class type is used as our persistent class.

Transport

With the TDSServer and TDSServerClass components in place, it's time to add the transport layer. DataSnap 2010 offers two choices here: HTTP or TCP/IP.

The HTTP transport protocol is implemented by the TDSHTTPService component. This component has an HttpPort property, by default set to port 80, that you may need to change (especially if you have a web server already running on your development machine). Apart from that, the TDSHTTPService component must explicitly be "started" (and stopped) using the Active property. You may encounter some funny behavior if you set Active to true at designtime and then also run the application (which results in two instances listening to the same port). This is another good reason not to set this property to Active at designtime, but to instead change the property via code (e.g., in the constructor of the form).

The TDSHTTPService component also has a help-

ful AuthenticationManager property that we can connect to the TDSHTTPServiceAuthenticationManager component. This will enforce the use of HTTP Authentication (which can be verified in the OnHttpAuthenticate event handler of the TDSHTTPServiceAuthenticationManager component). Warning: since the protocol itself is still HTTP, the HTTP Authentication information (username and password) are being sent in plain text, so anyone with a packet sniffer can read them. It would be better to use the HTTPS protocol in combination with HTTP Authentication, but unfortunately, DataSnap 2010 does not support HTTPS just yet (see QC #81335 for more details on this missing feature).

In addition to HTTP, DataSnap also supports TCP/IP, and this is implemented by the TDSTCPServerTransport component. This one does not come with built-in authentication, but we can always implement the OnConnect event handler of the TDSServer component and check the values of the DSAuthenticationUser and DSAuthenticationPassword ConnectProperties values manually (the same property values are used for the automatic HTTP Authentication check), but that's a story for another day.

To continue our demo, place a TDSTCPServerTransport component on the form, and point the Server property to the TDSServer component. We can configure this component, specifically we should probably change the default port by modifying the Port property from the default 211 and use another value. Make sure to remember that value, since the client needs to specify the new port as well, of course.

Compile and run the DS2010BCBServer project (communicating using TCP/IP over the default port 211). If a Windows Firewall warning appears, allow the program to communicate on the private network.

That's it for the server. Now you can build DataSnap 2010 clients, connecting using TCP/IP over port 211, using DSProviderConnection to get to the DataSetProvider exported from the DSDataModule

DataSnap client

We can now add a DataSnap Client application, typically by adding it to the same project group. Right-click on the project group and do "Add New Project | VCL Forms Application." I've saved the form in MainForm.cpp and the project in DSClient. cbproj.

In order to connect to the (running!) DataSnap server, we should place a TSQLConnection component on the form, set the Driver property to DataSnap and the driver subproperties should by default be set to communicate using TCP/IP over port 211. Note that we may need to change that port number if we modified it at the server side. Apart from that, all we need is to set the LoginPrompt property to false and the Connected property to true to see if we can connect to the server.

When this is verified, we can add another component, this time from the DataSnap Client category on the Tool Palette (a category which mainly hosts older DataSnap Client components, by the way). We need to place a TDSProviderConnection component on the form, and connect its SQLConnection property to the TSQLConnection component. Next we need to specify the value for the ServerClassName. This was the type name of the data module at the server side—the type that was used for our Persistent Class as assigned in the OnGetClass event of the TDSServerClass, remember? The actual type is TDSDataModule, and we need to enter that as the value for the ServerClassName property. It would have been nice to show a drop-down list of all exported server class names, but alas that's not the case.

The next step involves placing a TClientDataSet, connecting its RemoteServer property to the TDSProviderConnection component. Then, if all previous connections were made correctly (and the ServerClassName property of the TDSProviderConnection component is also correct), we can open up the ProviderName property of the TClientDataSet, and this will show a list of exported names of TDataSetProvider components from the server module. In this case, we should see only dspEmployees. If you do not see the choice dspEmployees, you need to verify the previous steps for the DataSnap Client, and also make sure the DataSnap Server itself is running (and not blocked by the firewall).

Finally, we can add a TDataSource, TDBGrid, TDBNavigator and other data-aware controls you want to use to display the data from the DataSnap Server in the thin (or smart) client. If we set the Active property of the TClientDataSet to true, we will see live data at designtime in the DataSnap Client as shown in **Figure 3**.

Warning: Make sure to set the Active property of the TClientDataSet back to false when you save and close the project. And also ensure the Connected property of the TSQLConnection component is set to

false. The reason will become obvious once you open the project group again: by default the main form of the last project will be shown, but if these components are "active", then they will try to make a connection to the DataSnap Server. The server is the first project in the group, but usually not running when you just open the project group. So, this will lead to a certain period in which C++Builder may be frozen after you get a connection timeout. To avoid that problem, I always ensure that `Active` and `Connected` are set to false before saving and closing the project.

DataSnap server methods

Apart from exporting `TDataSetProviders` from the server side to the clients, DataSnap 2010 also offers the support of so-called server methods. These are methods that we can define as public functions in the `TDSServerMethods` class, and who will be exported to the outside world as well—in theory that is, since this doesn't work quite out-of-the-box for C++Builder. The problem is that this exporting of server methods relies on so-called RTTI Method Information, which is currently only supported for Delphi with a `{$METHODINFO ON}` compiler directive.

There are two workarounds for this problem. In this article, I will show you one, which is based on the fact that C++Builder can compile Delphi code. But it's not a pure C++ solution. If you want the pure C++Builder solution, without Delphi code, you may want to check out Hanno Nagland's C++Builder helper functions and macros at [2].

Delphi server methods

Like I said, I will stick to showing you how to use C++Builder to compile Delphi code. For that, we need a .pas and .dfm file, which you may not be able to create if you have C++Builder 2010 without the full RAD Studio 2010 (which also includes the Delphi personality). Assuming you only have C++Builder, you need to create a file ServerModule.dfm with the following contents:

```
object BCBServerModule: TBCBServerModule
  OldCreateOrder = False
  Height = 480
  Width = 640
end
```

And a file ServerModule.pas with the following lines of Delphi code:

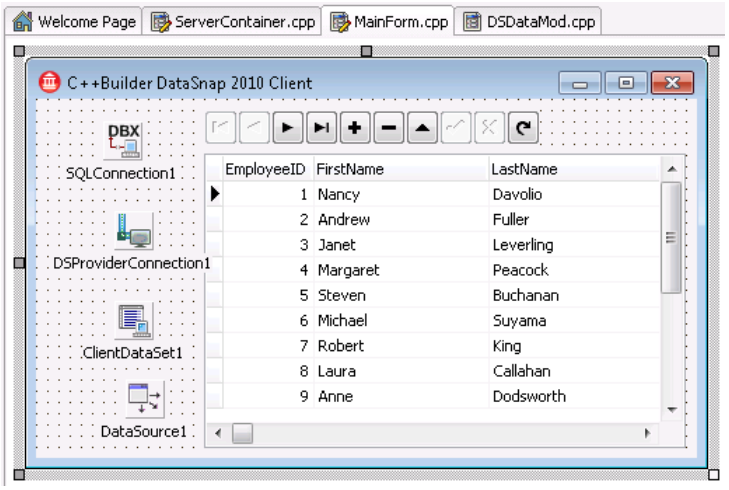


Figure 3: *Live data at designtime.*

```
unit ServerModule;
interface
uses
  DSServer;

type
  TBCBServerModule = class(TDSServerModule)
  private
    { Private declarations }
  public
    { Public declarations }
    function EchoString(const S: String):
      String;
  end;

implementation
{$R *.dfm}

{ TBCBServerModule }

function TBCBServerModule.EchoString(
  const S: String): String;
begin
  Result := S;
end;

end.
```

This is just an empty Server Module, but we can copy the `TSQLConnection`, `TSQLDataSet` and `TDataSetProvider` from our `TDSDataMod` to the `TDSServerModule` if you want.

Adding server methods can now be done in the Delphi unit, using Delphi syntax. As an example, the above unit already includes the `EchoString` function. You can add more public server methods, and the `{$METHODINFO ON}`—defined for the parent class `TDSServerModule`—will ensure that these server methods will be exported from the DataSnap server

when the .pas file is complied.

In both cases, we should ensure that the TDSServerClass component points to the right server module type. In my Delphi server module example, the OnGetClass event handler should be changed as follows:

```
void __fastcall TDSForm::
  DSServerClass1GetClass(
    TDSServerClass *DSServerClass,
    TPersistentClass &PersistentClass)
{
  //PersistentClass =
  //__classid(TDSDataModule);
  PersistentClass =
    __classid(TBCBServerModule); // Delphi
}
```

This code will ensure that we use the TBCBServerModule that exposes the EchoString function with RTTI method info attached.

Feel free to add any other custom additional server methods, but remember they have to be implemented in Delphi (unless you use the C++Builder helper functions and macros from Hanno Nagland).

Server methods client (in C++)

Apart from defining server methods, we should also be able to call them. Fortunately, this is where we can use pure C++ again.

Assuming you have exported DataSnap server methods—either using the Delphi syntax, or by using the C++Builder help functions and macros from Hanno Nagland—we can return to the DataSnap client project in C++Builder. Make sure the DataSnap server is running, and then right-click on the TSQLConnection component on the DataSnap client form. Select the option "Generate DataSnap client classes", as shown in **Figure 4**.

The result is a generated C++Builder proxy class of the server methods, with the following definition of the TBCBServerModuleClient class.

```
class TBCBServerModuleClient :
  public TObject
{
private:
  TDBXConnection *FDBXConnection;
  bool FInstanceOwner;
  TDBXCommand *FEchoStringCommand;

public:
  __fastcall TBCBServerModuleClient(
  TDBXConnection *ADBXConnection);
```

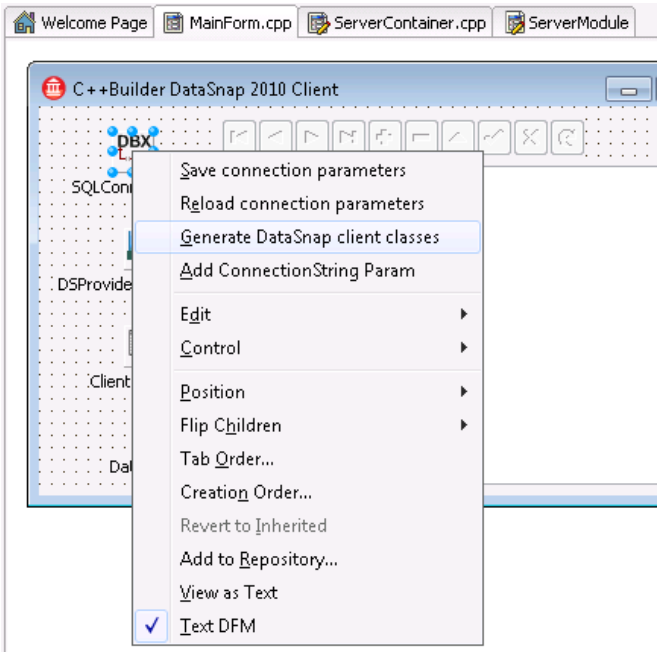


Figure 4: *Context menu option to generate client classes.*

```
  __fastcall TBCBServerModuleClient(
  TDBXConnection *ADBXConnection,
  bool AInstanceOwner);
  __fastcall ~TBCBServerModuleClient();

  System::UnicodeString __fastcall
    EchoString(System::UnicodeString S);
};
```

Two constructors, one destructor and our single server method called EchoString. We can call the constructor by passing the DBXConnection property of the TSQLConnection component as argument, and then we can call the EchoString method. In pure C++Builder code, this would be as follows:

```
void __fastcall TForm1::
  btnEchoStringClick(TObject *Sender)
{
  SQLConnection1->Open();
  TBCBServerModuleClient* WS =
    new TBCBServerModuleClient(
      SQLConnection1->DBXConnection);
  ShowMessage(WS->EchoString(
    "Called from C++ Client"));
  SQLConnection1->Close();
}
```

Although the server side it not yet perfect (out-of-the-box), I can only encourage people to vote for the C++Builder feature requests in Quality Central such as QC #83542 (for the missing DataSnap Wizards).

DataSnap deployment

The "old" DataSnap servers, based on COM, would automatically register themselves in the old days (using C++Builder 6 to 2006). This protocol was changed with C++Builder 2007, since Windows Vista would complain if an application tried to register itself while not being run "as administrator".

With DataSnap 2010, there is no more COM, and no need (or way) to register the location of the DataSnap server. An incoming client connection can no longer automatically cause the server to be started. In other words: when deploying a DataSnap server application, we should ensure that the server application is always up-and-running to allow the client(s) to connect to it.

A regular VCL Forms application may not be the best server type, which is why you may want to turn your DataSnap server into a Windows Service Application, or into a WebBroker (ISAPI) project. If you do the latter, then you can use the web module to place your DataSnap server components (that are currently residing on the main form). Both a Windows Service and a Web Server application will be available without the need for someone to logon to the server machine (and starting the DataSnap server application).

Summary

In this article, I've shown how we can use C++Builder 2010 to build DataSnap 2010 server and client applications. Although some support is still missing—like the DataSnap Wizards and direct C++Builder support for server methods—we can get functional servers and clients.

Hopefully, Embarcadero will add the missing functionality of DataSnap 2010 in future releases or updates of C++Builder.

Contact Bob at **Bob@eBob42.com**

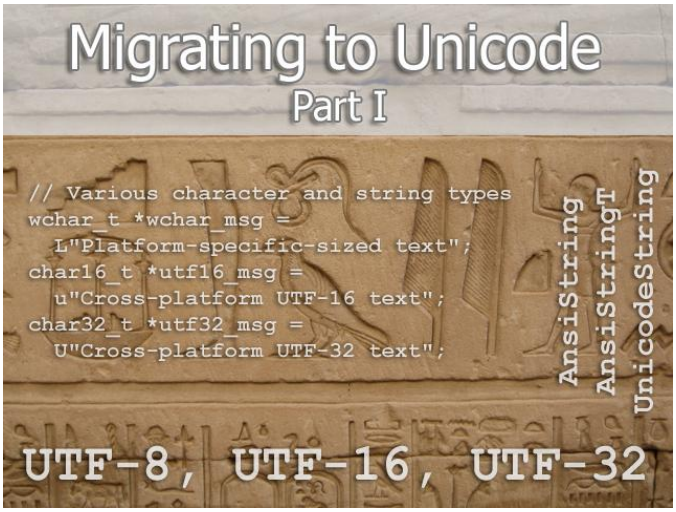
References

1. B. Swart, "Database Development with C++Builder, Part VI: DataSnap," *C++Builder Dev. Journal*, **9** (10), 2005.
2. http://cc.embarcadero.com/item/27643

Migrating to Unicode, Part I

By Josh Kelley

Versions: C++Builder 2010, 2009



One of the biggest changes in C++Builder 2009 and 2010 is the addition of full Unicode support throughout the VCL and RTL. Unicode support is a major step forward for the VCL and is critical for internationalization, but its implementation as an absolute requirement within the VCL can seem like a major obstacle in upgrading to C++Builder 2009 or 2010.

Fortunately, migrating to Unicode can be much less daunting than it first appears. The first key realization in migrating to Unicode for C++Builder 2009 or 2010 is this: You do not have to migrate to Unicode to use C++Builder 2009 or 2010. C++ is a diverse language, permitting the use of many libraries and several programming paradigms, and while your code that uses the VCL needs to be Unicode-aware, your code that uses the C runtime library, the STL, or the Windows API (for example) can continue to use ANSI and only convert to Unicode when passing data to or from the VCL. A complete migration to Unicode is obviously necessary to gain the full benefits of internationalization, but migrating only the VCL portion of your code can drastically simplify the task of upgrading to C++Builder 2009 or 2010 while letting you gain the other significant benefits that those versions offer (such as Boost, C++0x support, gestures, and a Ribbon control).

Whether you choose to make your entire application Unicode-aware or to upgrade only the portions that interact with the VCL, there are several C and C++ development techniques that can make the task much easier. Part I of this article offers an introduction to Unicode and discusses working with Unicode in C, C++, and the VCL, while Part II examines some of these Unicode migration techniques in more detail.

Remember, though, that supporting Unicode is only part of internationalizing an application. Full internationalization also includes issues such as date and time format, sorting orders, support for right-to-left layout, culture-neutral icon design, and so on. Such issues are beyond the scope of this article.

An introduction to Unicode

This is a brief introduction to Unicode. For a more thorough background, see [1] or [2].

Unicode is the international standard for representing text from almost any language in a computer. Unicode was designed to replace the older ANSI and ASCII standards and to address those standards' disadvantages in dealing with international text.

ASCII (such as was used on the original IBM PCs) was designed for plain English text only. It represented each character as a number between 32 and 127. Space was 32, '0' was 48, 'A' was 65, and so on. ASCII characters were stored one per byte, but since ASCII only defined characters through 127, byte values 128-255 were assigned a number of different meanings depending on where they were used. (The original IBM PC assigned various accented characters and line drawing characters to the 128-255 range; later, various countries assigned letters from their own languages' alphabets; and so on.)

The ANSI standard kept characters 32-127 the same as ASCII but standardized the use of the 128-255 range into a series of *code pages*. Each language or region could be assigned its own code page, and as long as 8-bit textual data was associated with a code page, it could faithfully represent non-plain-English text. ("ANSI" is actually a bit of a misnomer; Windows' code pages were never standardized by ANSI.)

The Unicode standard was introduced to cover two major shortcomings with the ANSI standard. First, the ANSI standard made no provision for multilingual text that needed more than one code page. Second, alphabets such as Chinese have thousands of characters and so cannot fit in a single code page.

Unicode provides a standard way of referring to any of over 100,000 characters ([2]). Each of these 100,000 unique characters (called *code points* in Unicode terminology) is assigned a unique name and a unique numeric identifier, which is written as "U+" followed by a 4-digit hexadecimal value. For example, the code point for the English capital A is named "LATIN CAPITAL LETTER A" and is written U+0041. It is important to note that Unicode characters *are not guaranteed to be representable by a 16-bit value*. The portion of Unicode characters that *can* be represented in 16 bits is called the *Basic Multilingual Plane*. Characters outside of the Basic Multilingual Plane are primarily used for ancient scripts (such as Egyptian hieroglyphics), musical notation, and rarely used Han ideograms.

Because code points are abstract entities, Unicode provides several *encodings* to represent these abstract code point values in a machine-readable form. An encoding describes how to represent each *code point* as one or more *code units*. (A code unit is simply "the minimal bit combination that can represent a unit of encoded text" [3].) The most common encodings are UTF-8, UTF-16, and UTF-32.

- UTF-32 uses 32-bit code units. UTF-32 has the advantage that each code point takes a consistent amount of space, but it tends to be wasteful of space (since English characters can otherwise be represented in 8 bits, and most characters in use can otherwise be represented in 16 bits). Because of its high storage requirements, UTF-32 is rarely used.
- UTF-16 uses 16-bit code units. UTF-16 can represent *most* of the code points in use as a single code unit. Code points above U+FFFF must be represented using *surrogate pairs*, a pair of code units that together represent a single code point. UTF-16 has the advantage that *most* code points can be represented as single code units and thus take a consistent amount of space; however, the storage requirements are higher than UTF-8, and the possibility of surrogate pairs is easy to overlook in development and testing. UTF-16 is the preferred encoding for a number of platforms and libraries (including Windows, OS X, Java, and .NET).
- UTF-8 uses 8-bit code units. Byte values between 32 and 127 are identical to ASCII, and UTF-8 strings can use a terminating NULL character just like standard C ASCII strings. Code points above 127 are represented by sequences of up to 4 bytes. UTF-8 has the advantage of being backwards compatible with ASCII; it has the disadvantage that code units frequently take varying amounts of space (and so operations like "take the 100th character from this string" can become much harder). Because UTF-8 has lower space requirements, it is commonly used for web pages, email, and similar stored or transmitted data. It is also the preferred encoding for a few platforms and libraries (including the Linux kernel and the GTK framework).

An additional complication of Unicode is the existence of composite (or composable) characters. A letter with accent marks or diacritical marks can be represented in Unicode either as a unique, precomposed code point or as the code point for the plain letter followed by the code point for its diacritical mark (or possibly multiple code points for multiple diacritical marks). For example, é can be represented in Unicode either as U+00E9 ("Latin small letter e with acute") or as U+0065 ("Latin small letter e") followed by U+0301 ("combining acute").

The existence of composable characters means that a simple binary comparison is not sufficient for checking two Unicode strings for equality. Because of composable characters, UTF-16's surrogate pairs, and UTF-8's variable length encodings, it is no longer valid to assume that accessing arbitrary characters out of a string or splitting string at arbitrary indexes will work. Operating system APIs and third-party libraries such as ICU [4] offer routines to help with these complications. (Windows APIs in particular will be discussed below under "Unicode in the Windows API.")

Working with Unicode

Unlike languages which offer a single built-in string data type, C++Builder offers several choices: code can use C-style characters and strings, or C++ `string` objects, or VCL `String` objects. Each of these has its own

set of Unicode variations. Similarly, the Windows API provides both ANSI and Unicode variants.

Unicode data in C

A C-style string is simply an array of `char` values, terminated by a NULL byte (also written as '`\0`'). Each `char` takes one byte of storage. (This is guaranteed by the C and C++ standards; as a pedantic note, however, one byte of storage is not guaranteed to be 8 bits, and some rare platforms use 16 bits or other sizes.) The encoding of a `char` string is not specified; it could be straight ASCII, or any of the ANSI code pages, or even UTF-8, although in Windows, it's generally assumed to be in the system default ANSI code page. Working with Unicode introduces several more C and C++ data types for C-style strings:

- `wchar_t`: C and C++ apps have traditionally used `wchar_t` as a replacement for `char` when working with Unicode strings. `wchar_t` strings are written as `L"Hello, world! \u263A"`. The size of a `wchar_t` is *compiler-dependent*: on Windows, it's 16 bits and assumed to contain UTF-16 data; but on Linux, it's 32-bits; and other platforms may use values as small as 8 bits. If you need truly cross-platform Unicode-aware code, you may need to avoid the built-in types altogether and instead use a third-party library such as ICU [4].
- `char16_t`, `char32_t`: C++0x (the draft of the new standard for the C++ language) specifies these two new character types for holding UTF-16 and UTF-32 data, respectively. `char16_t` values are written as `u"Hello, world! \u263A"`. `char32_t` values are written as `U"Hello, world! \u263A"`. (C++0x also allows using `u8"Hello, world! \u263A"` to represent UTF-8 data as a `char` array, but C++Builder doesn't support this.)

Of course, having Unicode data types does little good if you have no way to specify some of the more esoteric Unicode characters. Traditional C strings can represent nonprintable characters using predefined escape characters like \n (newline) as well as arbitrary hexadecimal values like \x7f. Similarly, Unicode characters can be written as \u followed by their 4-digit hexadecimal value. (For example, `L"\u00E9"` is a "Latin small letter e with acute," and `L"\u263A"` is a smiley face.)

Because the C++Builder IDE is fully Unicode-aware, you can also directly enter Unicode characters into your code, without resorting to escape characters. There are a few ways to enter Unicode characters in Windows: for example, you can use Windows' built-in Character Map utility, or you can hold the Alt key while typing '+' followed by the Unicode character's 4-digit hexadecimal value. The fileformat.info web site has a complete list of input methods for Windows [5] as well as a searchable database of Unicode characters [6]. C++Builder automatically uses the UTF-8 encoding for source files containing Unicode characters.

Sprinkling Unicode escape characters throughout your code hampers readability, and directly entering Unicode characters can sometimes be difficult to work with. An alternative is to use preprocessor `#defines` to create macros for Unicode characters which your application needs. Using preprocessor macros instead of `const` values is often discouraged in modern C++ development, but using `#defines` for Unicode and other string constants have the advantage that they can be automatically concatenated, at compile time, without having to clutter your code with concatenation operators.

Listing 1: *Unicode in C*

```
// Various character and string types
wchar_t *wchar_msg =
  L"Platform-specific-sized text (UTF-16 on Windows)";
char16_t *utf16_msg =
  u"Cross-platform UTF-16 text, new with C++0x";
char32_t *utf32_msg =
  U"Cross-platform UTF-32 text, new with C++0x";
#if 0
char *utf8_msg =
  u8"UTF-8 text, unsupported by C++Builder";
#endif

// A smiley face as a Unicode escape code
wchar_t *msg1 = L"Hello, world! \u263A\n";

// A smiley face using a preprocessor macro and string
// concatenation.
#define SMILEY_FACE L"\u263A"
wchar_t *msg2 =
  L"Hello, world! " SMILEY_FACE "\n";
// In a real project, such #defines would probably go
// in their own project-wide header file.
```

Listing 1 shows examples of Unicode literals, escape codes, and preprocessor macros.

Working with Unicode in C

C developers are used to using <string.h> functions such as `strlen()`, `strcpy()`, and `strcat()` to manipulate C-style strings. There are corresponding functions for manipulating C-style `wchar_t` strings; most `wchar_t` functions are defined both in <wchar.h> and in the "traditional" header file (<string.h> for plain string manipulation, <stdio.h> for I/O, etc.).

- For `wchar_t` string manipulation, use `wcslen()`, `wcscpy()`, `wcscat()`, and so on. (Replace "str" with "wcs.")
- For `wchar_t` file I/O, use functions like `fgetws()` and `fputwc()` instead of `fgets()` and `fputc()`. (Insert "w" before the data type.)
- `printf()`, `sscanf()`, and so on become `wprintf()`, `swscanf()`, and so on. (Insert "w" before "printf" or "scanf.") Take note to not confuse `swprintf()` (wide character `sprintf()`) with `wsprintf()` (the Windows implementation of `sprintf()`).
- File and directory manipulation functions, such as `fopen()`, `opendir()`, `mkdir()`, and `_unlink()`, become `_wfopen()`, `_wopendir()`, `_wmkdir()`, and `_wunlink()`. (Add "_w" to the beginning.) These let you manipulate files and directories with Unicode characters in their names.

The `printf()` and `scanf()` family deserve special mention. `wprintf()` and `wscanf()` act like their narrow character counterparts in most respects, but the format strings needed to specify char and char * arguments changes for wide characters (and also differs between compilers). For example, Visual C++ and C++Builder's implementations of `wprintf()` and `wscanf()` interpret `"%s"` as `wchar_t`, while GCC's runtime library on Linux interprets `"%s"` as `char` (just like `printf()` and `sscanf()`). (For vender references and the C99 draft standard's take on this, see [7], [8], and [9].)

Table 1 summarizes how format specifiers are interpreted for different platforms. The fact that C++Builder's treatment of format specifiers depends on the function being called, combined with this inconsistency in how specifiers are handled between compilers, can be a source of confusion, so be careful.

Platform	`"%c"` or `"%s"`	`"%lc"` or `"%ls"`	`"%hc"` or `"%hs"`
C99	`char`	`wchar_t`	undefined
Windows (VC++ and C++Builder)	`wchar_t`	`wchar_t`	`char`
Linux (GCC)	`char`	`wchar_t`	`char`

Table 1: *Format specifiers for wprintf and wscanf.*

These C string `wchar_t` functions are inconsistently documented; the wchar.h topic in the RAD Studio 2009 Documentation (which is more conveniently organized than RAD Studio 2010's documentation in this case) has a semi-complete listing, or you can browse the include files yourself.

Unicode in C++

String manipulation in C++ generally involves the use of the `std::string` class, as well as the various <iostream> classes for input, output, and string buffering. Developers familiar with Boost may also uses classes such as `boost::regex` or `boost::format` to help with string manipulation.

As it turns out, switching to the `wchar_t` version of these classes is quite easy: just prefix a w to each

Listing 2: *Unicode in C++*

```
#include <tchar.h>
#include <string>
#include <iostream>
#include <sstream>

using namespace std;

int _tmain(int, _TCHAR*)
{
  int value;

  // ANSI (narrow character) code
  string s1 = "123";
  stringstream stream1(s1);
  stream1 >> value;
  cout << "The value is " << value << endl;

  // Wide character version. Easy!
  wstring s2 = L"124";
  wstringstream stream2(s2);
  stream2 >> value;
  wcout << L"The value is " << value << endl;

  return 0;
}
```

class name. See **Listing 2** for sample char string code in C++ and its corresponding wchar_t code.

Most text-related classes (including <iostream>) in the C++ Standard Library and in Boost are actually typedefs for template classes. For example, std::string is actually a typedef for std::basic_string<char>, and std::wstring is a typedef for std::basic_string<wchar_t>. Because std::basic_string is a template, it can be instantiated on any char-like data type that you wish. This means that, if you need to work with C++0x's char16_t or char32_t data types, you can use std::basic_string<char16_t> instead of std::string, use std::basic_fstream<char32_t> instead of std::fstream, and so on.

Unicode in the VCL

This is where things get interesting. Starting with RAD Studio 2009, the VCL offers several string classes which support ANSI, UTF-8, and UTF-16 encodings:

- AnsiString corresponds to the old String class. It contains 8-bit (char) data in the system default code page.
- UnicodeString is the new class, containing 16-bit (wchar_t) data in the UTF-16 encoding.
- WideString still exists from previous versions of RAD Studio. It corresponds to COM's BSTR data type and contains 16-bit (wchar_t) data like UnicodeString. Because UnicodeString uses C++Builder's own memory management and reference counting, it's often faster than WideString, so unless you need easy interoperability with COM, you should use the new UnicodeString class.
- AnsiStringT is a class template that contains 8-bit (char) data encoded in *any* code page; the code page is given as the template parameter. (AnsiString is actually a typedef for AnsiStringT<0>.) The requirement that the code page be given as a template parameter prevents you from using AnsiStringT with arbitrary code pages at runtime, so if you need that capability, you may need to instead use RawByteString (below) or use one of the C or C++ string manipulation methods instead of using the VCL.
- UTF8String is an AnsiStringT instantiation using the UTF-8 encoding.
- RawByteString contains 8-bit (char) data in an *unspecified* code page. The VCL will avoid applying any code page conversions to RawByteStrings; it becomes the calling code's responsibility to correctly handle code pages issues. Using RawByteString can have several advantages: since each code page is otherwise a separate compile-time type, RawByteString lets you write a single routine that can handle any code page; it removes any VCL overhead of doing code page conversions itself; and it prevents possible loss of data from automatically converting text data into encodings that can't represent some characters.

Most member functions of these new string classes operate just the same as they did for the old pre-C++Builder 2009 String class. The printf-type methods (printf(), sprintf(), vprintf(), cat_printf(), cat_sprintf(), and cat_sprintf()) deserve special mention. Like C's wprintf() and wscanf() functions, their treatment of the "%s" and "%c" format specifiers depends on whether they're called on an AnsiString or UnicodeString instance. Refer back to **Table 1** for details.

Unicode in the Windows API

The Windows API includes both Unicode and ANSI variants. For example, the MessageBox function is actually two different functions: MessageBoxA, which takes ANSI strings, and MessageBoxW, which takes wide (UTF-16) strings. MessageBox itself is a macro that resolves to MessageBoxA or MessageBoxW depending on your preprocessor macros and project options. You're also free to explicitly call one API variant or the other, regardless of your project options, simply by calling MessageBoxA or MessageBoxW directly. Other Windows API functions dealing with text or string data have similar variants.

Which variant you get is determined by whether or not the UNICODE preprocessor macro is defined. In C++Builder, this macro is automatically defined depending on your project's options. To change this option from the IDE, go under the Project menu, under Options, under the top-level Directories and Conditions category, and examine the "_TCHAR maps to" option. If it's set to "char," then the UNICODE preprocessor macro is left undefined and the ANSI variant of the Windows API is used. If it's set to "wchar_t," then the UNICODE macro is defined and the wide-string

(UTF-16) variant of the Windows API is used.

The `UNICODE` macro also affects the use of tchar.h in writing code that can compile as ANSI or Unicode. This will be discussed in Part II.

The Windows API also includes functions such as `CharNext()`, `CharPrev()`, and `CompareString()` that are capable of dealing with complexities such as composite characters and surrogate pairs. See MSDN [10] for details.

Conclusion

In this article, I provided an introduction to Unicode and I presented an overview of how to use Unicode in C, C++, the VCL, and the Windows API. For a more detailed introduction to Unicode, see [11].

In Part II of this series, I'll show you how to migrate your existing C++Builder applications to use Unicode.

Contact Josh at **joshkel@gmail.com**.

References

1. Joel Spolsky, "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)." http://www.joelonsoftware.com/articles/Unicode.html
2. Wikipedia, "Unicode." http://en.wikipedia.org/wiki/Unicode
3. The Unicode Consortium. "Glossary." http://unicode.org/glossary/
4. "ICU – International Components for Unicode." http://site.icu-project.org/
5. "How to enter Unicode characters in Microsoft Windows." http://www.fileformat.info/tip/microsoft/enter_unicode.htm
6. "Unicode." http://www.fileformat.info/info/unicode/index.htm
7. "MSDN: Size and Distance Specification." http://msdn.microsoft.com/en-us/library/tcxf1dw6%28v=VS.80%29.aspx.
8. "Linux Programmer's Manual: printf(3)." http://www.kernel.org/doc/man-pages/online/pages/man3/printf.3.html. Retrieved 4/12/2010.
9. "WG14/N1124 Committee Draft." http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf. Retrieved 4/12/2010.
10. "MSDN: String Reference: Functions." http://msdn.microsoft.com/en-us/library/ff468910%28VS.85%29.aspx. Retrieved 4/14/2010.
11. "Internationalization for Windows Applications (Windows)." http://msdn.microsoft.com/en-us/library/dd318661%28VS.85%29.aspx.

Get over 12 years of the Journal on CD!

Version 4.0 of our popular **Archive CD** contains over 12 years of the C++Builder Developer's Journal (from June 1997 through December 2008), all neatly presented through an easy-to-use HTML user interface.

You can view the CD contents with any browser and PDF reader.

Includes all article **source code** too!

With **hundreds of articles**, **illustrations,** and **source code** examples, this is the most complete set of information about C++Builder that you can find in one place!

Order: To order online, visit http://bcbjournal.org/archive_cd.htm

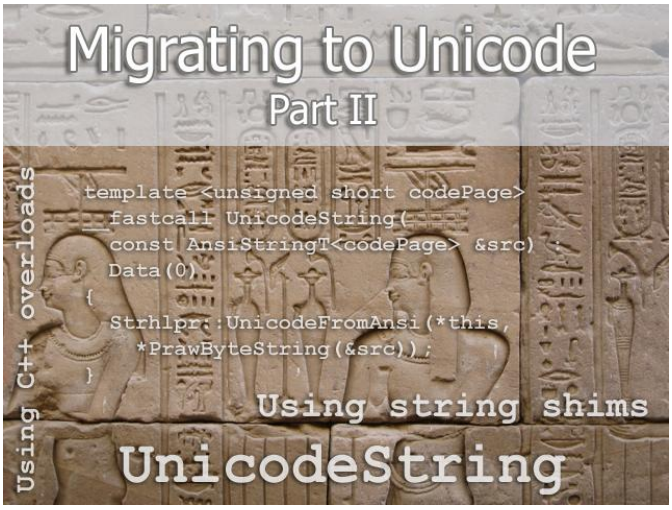
Price: Each CD is **$39.00** plus shipping.

Package: For just **$79.00**, get both a CD and a one-year subscription.

Migrating to Unicode, Part II

By Josh Kelley

Versions: C++Builder 2010, 2009



Part I introduced Unicode and covered the various options for working with Unicode text in C, C++, the Windows API, and the VCL. Now, in Part II, I will specifically discuss how to migrate a C++Builder application to Unicode.

Migrating C++Builder applications to Unicode

There are two basic approaches to handling Unicode issues when migrating a pre-2009 C++Builder application to C++Builder 2009 and above.

1. You can do a complete Unicode migration. Use the Windows Unicode APIs instead of ANSI APIs, replace char with `wchar_t`, replace `std::string` with `std::wstring`, and use `UnicodeString` instead of `AnsiString`. Depending on how your C and C++ code is written, this could be a major undertaking.
2. You can leave your application using ANSI and convert to Unicode only where it's absolutely necessary. Because only the VCL portion of C++Builder *requires* the use of Unicode, your C and C++ string manipulation and Windows API interaction can continue to use `char` and `std::string`. Even the portions of your code that interact with the VCL can often get away with continuing to use ANSI strings, thanks to the implicit conversions between `AnsiString` and `UnicodeString` that the C++ VCL provides (described in more detail below).

Of course, these two approaches aren't mutually exclusive. You can do an initial, "quick-and-dirty" migration using the minimal approach, and then gradually implement the complete approach for the portions of your application that would most benefit from Unicode support. And, even a completely migrated application may still need to deal with ANSI or UTF-8 when interfacing with legacy file formats or APIs, or when reading or writing data from or to the disk or network.

Regardless of which approach you choose, there are a number of C and C++ techniques that can be used to help with a Unicode migration:

- The Windows API includes functions for converting between ANSI and Unicode, and the VCL provides conversion constructors to easily convert between `AnsiString` and `UnicodeString` values.
- The standard Windows header tchar.h includes macros designed to let you write code that compiles as either ANSI or Unicode. This can help when converting code one portion at a time.
- C++-specific `typedefs`, and the use of C++ features such as function overloading, can fill in the gaps left by tchar.h in writing code that compiles as either ANSI or Unicode.
- The C++ concept known as "shims" (as described in Matthew Wilson's *Imperfect C++* [1] and as used in the STLsoft library [2]), combined with the use of C++ templates, can make it simple to write generic code that works with both `AnsiString` and `UnicodeString` (and C-style strings, and `std::string`, and anything else you care to support).
- The use of variadic functions such as `printf()` and `sprint()` presents a special challenge for migrating to Unicode, since the compiler is unable to catch ANSI-versus-Unicode issues with these.

Standalone scripts can be used to transform these variadic function calls into a format that the compiler can check and then revert them to their normal format after all issues are addressed.

Converting text to and from Unicode

Before any migration can proceed, you need to know how to convert between the various Unicode encodings and the various ANSI encodings. The two easiest ways are using the Windows API and using the VCL.

The relevant Windows API functions are `WideCharToMultiByte()` [3], which, despite its name, converts from UTF-16 to the encoding of your choice (UTF-8 or any of the various ANSI encodings); and `MultiByteToWideChar()` [4], which converts from the encoding of your choice to UTF-16. MSDN has full documentation on using these functions.

Converting using the VCL is even easier. The VCL provides C++ *conversion constructors* – constructors that can be called with only a single argument – so that you can construct a `UnicodeString` from an `AnsiString` or `UTF8String`, or vice versa. Because C++ conversion constructors are implicitly invoked as needed, this also lets you provide an `AnsiString` wherever a `UnicodeString` is needed, or vice versa. (For example, this lets you assign a `UnicodeString` to an `AnsiString`.) See **Listing 1** for example code.

The ease with which conversions can be done in the VCL can have drawbacks. Because the assignment operators look just like regular assignment and the conversion constructors can be implicitly invoked, your code may be converting between ANSI and UTF-16 without your even being aware of it. This can add runtime overhead, but more importantly, it can result in loss of data when converting from UTF-16 to an ANSI encoding that cannot represent all of the Unicode characters. Delphi includes a compiler warning when this happens ("W1058: Implicit string cast with potential data loss from 'string' to 'AnsiString.'"), but C++Builder will silently accept it.

Ideally there would be an option to have the C++Builder compiler emit a warning any time these ANSI-Unicode conversions are implicitly invoked, but as far as I can tell, no such option exists. If having these functions implicitly invoked is a concern for you, then the only solution is to modify C++Builder's header files.

To do this, open the file "include\vcl\dstring.h" and find the following lines:

Listing 1: *Converting to and from Unicode*

```
// Sample C++ functions for doing
// Unicode<->ANSI conversions using the
// Windows API. Note the use of
// boost::scoped_array to dynamically
// allocate memory and automatically clean
// it up once we're done.

std::wstring AnsiToUnicode(const char *s)
{
  DWORD size = MultiByteToWideChar(CP_ACP,
    0, s, -1, NULL, 0);
  if (size == 0) {
    return std::wstring();
   }
  boost::scoped_array<wchar_t> buffer(
    new wchar_t[size]);
  MultiByteToWideChar(CP_ACP, 0, s, -1,
    buffer.get(), size);
   return std::wstring(buffer.get());
}

std::string UnicodeToAnsi(const wchar_t *s)
{
  DWORD size = WideCharToMultiByte(CP_ACP,
    0, s, -1, NULL, 0, NULL, NULL);
  if (size == 0) {
    return std::string();
  }
  boost::scoped_array<char> buffer(
    new char[size]);
  WideCharToMultiByte(CP_ACP, 0, s, -1,
    buffer.get(), size, NULL, NULL);
  return std::string(buffer.get());
}

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
  // Implicit Unicode-to-ANSI conversion:
  AnsiString s1 = L"Hello, world!";
  // Implicit ANSI-to-Unicode conversion:
  UnicodeString s2 = "Hello, world!";

  // This works without modification in
  // C++Builder 2009, even though Caption
  // is Unicode and s1 is ANSI.
  Label1->Caption = s1;
  // An implicit conversion from Unicode to
  // ANSI. NOTE: This could lose data.
  s1 = Label2->Caption;
  // Identical to the above, but explicit.
  s1 = AnsiString(Label2->Caption);

  // We can also use a temporary AnsiString
  // or UnicodeString to do Unicode<->ANSI
  // conversions.
  MessageBoxA(Handle,
    AnsiString(Label1->Caption).c_str(),
    "Demo", MB_OK);
}
```

```
  __fastcall AnsiStringT(
    const WideString &src) :
    AnsiStringBase(src, CP){}
__fastcall AnsiStringT(
    const UnicodeString &src) :
    AnsiStringBase(src, CP){}
```

Change them to the following:

```
explicit __fastcall AnsiStringT(
    const WideString &src) :
    AnsiStringBase(src, CP){}
explicit __fastcall AnsiStringT(
    const UnicodeString &src) :
    AnsiStringBase(src, CP){}
```

This will let you explicitly do a Unicode-to-ANSI conversion when needed without the possibility of losing data in implicit conversions; see the explicit conversion example in **Listing 1**.

If you want to disable implicit ANSI-to-Unicode conversions (which cannot result in data loss but can be a performance loss), open the file "include\vcl\ustring.h" and find the following lines:

```
template <unsigned short codePage>
__fastcall UnicodeString(
    const AnsiStringT<codePage> &src) :
    Data(0)
{
    Strhlpr::UnicodeFromAnsi(*this,
      *PrawByteString(&src));
}
```

Change them to the following:

```
template <unsigned short codePage>
explicit __fastcall UnicodeString(
    const AnsiStringT<codePage> &src) :
    Data(0)
{
    Strhlpr::UnicodeFromAnsi(*this,
      *PrawByteString(&src));
}
```

Note that modifying system headers like this may cause problems when installing RAD Studio updates. See [5] for details.

Using tchar.h

It can be valuable to write character-width-agnostic code that can compile as ANSI or Unicode.

If you're planning on a complete Unicode migration as part of an upgrade to C++Builder 2009 or 2010, character-width-agnostic code lets you prepare for the migration in your previous version of C++Builder without breaking compilation.

If portions of your C++ code base are cross-platform, you may want those portions to use wide characters (UTF-16) on Windows but narrow characters (ANSI or UTF-8) on other platforms.

Windows provides the tchar.h header file to help with this. Depending on whether the `_UNICODE` preprocessor macro is defined (which, as discussed earlier, is controlled by C++Builder's "_TCHAR maps to" option), tchar.h defines the following macros:

- `TCHAR`, which is defined as char for non-Unicode builds and `wchar_t` for Unicode builds
- `_T`, which is removed by the preprocessor for non-Unicode builds and is defined as `L` for Unicode builds. This means that you can write `_T("Hello, world!")` and have the preprocessor convert it to a `char` literal (`"Hello, world!"`) or `wchar_t` literal (`L"Hello, world!"`) as appropriate.
- `_tcscat()`, `_tcscpy()`, `_tcscmp()`, etc., which are defined as `strcat()`, `strcpy()`, `strcmp()`, etc. for non-Unicode builds and `wcscat()`, `wcscpy()`, `wcscmp()`, etc. for Unicode builds
- `_tprintf()`, `_tscanf()`, `_stprintf()`, etc., which are defined as `printf()`, `scanf()`, `printf()`, etc. for non-Unicode builds and `wprintf()`, `wscanf()`, `swprintf()`, etc. for Unicode builds
- `_fgettc()`, `_fgetts()`, `_fputtc()`, etc., which are defined as `fgetc()`, `fgets()`, `fputc()`, etc. for non-Unicode builds and `fgetwc()`, `fgetws()`, `fputwc()`, etc. for Unicode builds
- `_ttoi()`, `_ttof()`, `_tcstol()`, etc., which are defined as `atoi`, `atof()`, `strtol()`, etc. for non-Unicode builds and `_wtoi()`, `_wtof()`, `wcstol()`, etc. for Unicode builds.

Macros are also provided for file- and directory-manipulation functions (so that you can manipulate ANSI or Unicode filenames as appropriate). See include\tchar.h for a complete list of available macros.

Using these macros is very simple:

```
TCHAR buffer[100];
_tcscpy(buffer, _T("Hello"));
_tcscat(buffer, _T(" world!"));
```

Using C++ typedefs

Windows' tchar.h provides mappings only for C functions and types, but it's trivial to extend the same concept to C++:

```
#ifndef _UNICODE
typedef std::string tstring;
typedef std::fstream tfstream;
typedef boost::format tformat;
static std::istream& tcin(std::cin);
static std::ostream& tcout(std::cout);
#else
typedef std::wstring tstring;
typedef std::wfstream tfstream;
typedef boost::wformat tformat;
static std::wistream& tcin(std::wcin);
static std::wostream& tcout(std::wcout);
#endif
```

Extend as needed by creating `typedefs` for the Standard Library and Boost classes and references for the variables (such as `cout` and `cin`) which are actually used in your code.

Using ANSI and Unicode variants of the Windows API

As already discussed, Windows provides ANSI and Unicode variants of its API. For example, `MessageBox()` is actually a macro that's defined as `MessageBoxA()` (ANSI) or `MessageBoxW()` (Unicode), depending on your project settings. As part of your Unicode migration, you can explicitly call the ANSI or Unicode variant from different parts of your code, as different parts are migrated.

Using C++ overloads

Using tchar.h and the Windows API ANSI and Unicode variants, you can convert much of your existing code to be character-width-agnostic using only some search-and-replace operations in your IDE editor or in a script. If you want to avoid doing even this much, or if you have some third-party APIs that don't support both ANSI and Unicode, you can use C++ overloading to create your own functions that support both ANSI and Unicode. (Keep in mind that tchar.h was designed for C programs; as a C++ developer, you have more powerful tools available.)

For example, instead of replacing `strcpy()` with `_tcscpy()`, you could define a new function:

```
inline wchar_t *strcpy(wchar_t *dest,
  const wchar_t *src)
{
  return wcscpy(dest, src);
}
```

Include this function throughout your project, and repeat as needed for other functions. The overloads that you define may be able to call a wide or narrow character equivalent (as this example `strcpy()` overload does with `wcscpy()`), or they may need to do a Unicode-to-ANSI conversion themselves for APIs that have no Unicode support.

Overloading other libraries' functions in this manner can hamper the readability of your code. This technique should probably be viewed as a quick and temporary way of getting up and running in C++Builder 2009 and 2010 rather than as a long-term solution.

Special techniques for migrating to Unicode

Using string shims

The techniques discussed so far for calling the right functions with the right kind of text data all have various shortcomings. As an example, consider how to most easily migrate ANSI code that calls `TApplication::MessageBox()` for user feedback.

- The VCL offers convenient conversions between ANSI and Unicode, but having to use these at every call site can be tedious (especially if adding these conversions is a requirement for the initial migration to C++Builder 2009 or 2010) and can hamper readability. `TApplication::MessageBox()` takes `wchar_t*` parameters instead of `UnicodeStrings`, so it cannot benefit from implicit VCL ANSI-to-Unicode conversions. Neither can various non-VCL C and C++ APIs.
- Using macros, as tchar.h does, lets you select between ANSI and Unicode at compile time, but it doesn't let you use both within a single build, and it requires that you clutter your source code's namespace with extra macros.
- Creating C++ overloads for ANSI and Unicode variations is perhaps the easiest from the caller's perspective, but it requires that you create separate functions for every argument combination. For example, `TApplication::MessageBox()`

takes two arguments (a message and a caption), and so a `TApplication::MessageBox`-style function would need four overloads (ANSI message and caption; Unicode message and caption; ANSI message and Unicode caption; Unicode message and ANSI caption).

And this is only for a bare bones `TApplication::MessageBox`-style function. Most other VCL functions take `Strings`, not `wchar_t*`, as parameters; it would be convenient if we had overloads to do the same for our hypothetical `MessageBox()` replacement, but that adds even more overloads. It would be even more convenient if we could also support C++ Standard Library types like `std::string` or COM-related types like `WideString` or `BSTR`. The number of overloads to require all of these combinations of parameters for even a single function quickly becomes prohibitive. Clearly, a better approach is needed.

The concept of shims, as promoted by C++ author and developer Matthew Wilson, offers a solution. Shims "are small, lightweight (in most all cases having zero runtime cost) components that help types 'fit or align' into algorithms and client code" [6]. For example, suppose you had a function that, if given any string-like object, gave you a pointer to a C-style string. (Since this function gets a pointer to a wide C-style string, and following the convention of Matthew Wilson's STLSoft library, we'll call this function `c_str_ptr_w()`.) Then you could write the following `TApplication::MessageBox()` replacement:

```
template <typename T1, typename T2>
int AppMessageBox(const T1& Text,
  const T2& Caption, int Flags = MB_OK)
{
  return Application->MessageBox(
    c_str_ptr_w(Text),
    c_str_ptr_w(Caption), Flags);
}
```

Now we simply need to make sure that `c_str_ptr_w()` results in a valid argument for every parameter type that we use for `AppMessageBox()`. The first few parameter types are easy:

```
inline const wchar_t *c_str_ptr_w(
  const wchar_t *s)
{
  return s;
}

inline const wchar_t *c_str_ptr_w(
  const UnicodeString& s)
{
  return s.c_str();
}

inline const wchar_t *c_str_ptr_w(
  const std::wstring& s)
{
  return s.c_str();
}
```

So far we're following the practice described by Matthew Wilson in [6] and [1] and implemented in the STLSoft library [2]. We have a replacement for `TApplication::MessageBox()` that we can switch to with a simple search-and-replace (just replace "Application->MessageBox" with "AppMessageBox") and that take any of several types of Unicode arguments without excessive overloads or extra function calls.

For the purpose of quickly migrating to Unicode, however, it's useful to have a `TApplication::MessageBox()` replacement that can also take ANSI arguments. In his article on shims and in his work on the STLSoft library, Matthew Wilson explicitly avoids providing shims that convert between ANSI and Unicode, since those introduce (in his words) "semi-implicit" conversion operations that introduce a performance penalty and violate the expectation that shims be lightweight. However, as part of a C++Builder 2009 or 2010 Unicode migration, it's more useful to accept a (possibly negligible) performance penalty in order to complete the initial migration as soon as possible, then address performance and "proper" Unicode handling as needed.

Therefore, we need to provide `c_str_ptr_w` shims that take ANSI arguments (`const char *`, `AnsiString`, and `UnicodeString`). This is harder than the previous cases. Our code will have to take the following approach:

- We need to somehow provide a `const wchar_t` pointer. We can't simply return a `const wchar_t*` from `c_str_ptr_w()`, because we need to allocate memory to store the results of the ANSI-to-Unicode conversion, and returning a raw pointer to that allocated memory would constitute a memory leak.
- We *can*, however, define a class that contains the allocated memory and return a *copy of* (not a reference to nor a pointer to) that class. The C++ language guarantees that it will properly clean

up this temporary class instance in that case, and as long as we make everything inline, a good compiler will avoid the overhead of actually making the spurious copies of the class instance.

- Next, we define a C++ *conversion operator* that lets the class be implicitly converted to `const wchar_t*`.

The final code looks like this:

```
class c_str_ptr_w_string_proxy
{
public:
  explicit c_str_ptr_w_string_proxy(
    const char *s)
  : mString(s) {}
  // The above line does the actual ANSI-
  // to-Unicode conversion, using the VCL.
  // This is the C++ conversion operator:
  operator const wchar_t * () const
  {
    return mString.c_str();
  }

  // This is the buffer to store the
  // conversion:
  UnicodeString mString;
};

inline c_str_ptr_w_string_proxy
  c_str_ptr_w(const char *s)
{
   return c_str_ptr_w_string_proxy(s);
}

inline c_str_ptr_w_string_proxy
  c_str_ptr_w(const std::string& s)
{
  return
    c_str_ptr_w_string_proxy(s.c_str());
}
```

There are two caveats with using a shim of this sort: First, because the `wchar_t` pointer that's returned is to a temporary object, code such as the following results in undefined behavior:

```
void DoSomething(const AnsiString& s)
{
  const wchar_t *msg = c_str_ptr_w(s);
  // Invalid! msg's contents are undefined
  // at this point.
}
```

Second, the conversion operator is only invoked when the C++ compiler knows that it needs to be invoked. In particular, the compiler does not know to invoke it when it's used as part of a variadic function:

```
UnicodeString s;
// Dubious code; it pushes a copy of a
// c_str_ptr_w_proxy on the stack instead
// of calling the conversion operator.
s.sprintf("%s",
  c_str_ptr_w(some_ansi_string));
```

As it so happens, because of how `c_str_ptr_w_proxy` and `UnicodeString` are implemented, pushing a copy of a `c_str_ptr_w_proxy` on the stack behaves identically to calling the conversion operator, but it's a very bad idea to rely on implementation details like this. To avoid this problem:

1. Explicitly invoke the conversion operator when using a shim with a variadic function.
2. In C++Builder, turn on warning 8074 ("Structure passed by value") under Project, under Options, under C++ Compiler, under Warnings, to catch any dubious function calls such as this.

A complete set of appropriate shims functions is provided in the source code that accompanies this article.

Handling variadic functions

So now you're almost done. You've planned whether to do a complete or minimal migration. You've applied techniques such as tchar.h, C++ overloading, and string shims to ease the migration. You're using `RawByteString` and `UTF8String` where needed for dealing with external APIs or data. Everything compiles without warnings, and no immediate problems show when you run your application.

Unfortunately, one potentially major problem remains: variadic functions, primarily the `printf()` and `scanf()` family of functions (including the `printf()` family of methods on the VCL's `String` classes). Because these functions take variable arguments, the compiler does *no* checking on their arguments. The lack of type safety with these functions can be a recurring problem in C and C++ development but is a particular risk during a Unicode migration. For example, the following code will fail at runtime following a migration to C++Builder 2009 or 2010:

```
fprintf(logfile, "%s: Startup\n",
  Application->Title.c_str());
```

There are several approaches for providing type safety for variadic functions:

- Some compilers (such as GCC) analyze `printf()` and `scanf()` format strings at compile time and check arguments' types against the types specified by the format string. Unfortunately, C++Builder does not do this. Even compilers that do provide this feature often fail to check arguments for the wide character variations of `printf()` and `scanf()`.
- Variadic type safety issues can be avoided by switching to a C++ alternative to `printf()` and `scanf()`, which is `iostreams`, or one of the newer libraries such as Boost.Format. However, many developers find `printf`- and `scanf`-style format strings preferable to `iostream`'s syntax. Although libraries such as Boost.Format offer a good combination of `printf`-style format strings and C++ type safety, switching an entire codebase to a new formatting and I/O library just to complete a Unicode migration is not an option.

Nick Galbreath, in his blog posting "Type safe printf" [7], offers an intriguing alternative. He proposes a simple source code transformer that takes a C++ source file, searches for `printf`- and `scanf`-style functions, and transforms them. He gives the example of a program containing several bad arguments to `printf()`:

```
int main()
{
  int a = -1;
  printf("%d %u %hhd\n", a, a, a);

  unsigned int b = 512;
  printf("%d %u %hhd\n", b, b, b);

  double c = 2.0;
  printf("%d %f %u %hhd\n", c, c, c, c);

  return 0;
}
```

The type-safe `printf()` transformer turns each `printf()` call into its own unique function, whose arguments are specified as part of the function definition and therefore can be checked by the compiler:

```
/* VARARG TRANSFORMATION START */
/* This is autogenerated */

static void printf_1(const char* format,
    int a0, unsigned int a1, char a2) {
  printf("%d %u %hhd\n", a0, a1, a2);
}

static void printf_2(const char* format,
    int a0, unsigned int a1, char a2) {
  printf("%d %u %hhd\n", a0, a1, a2);
}

static void printf_3(const char* format,
    int a0, double a1, unsigned int a2,
    char a3) {
  printf("%d %f %u %hhd\n",
    a0, a1, a2, a3);
}
/* VARARG TRANSFORMATION END */

int main()
{
  int a = -1;
  printf_1("%d %u %hhd\n", a, a, a);

  unsigned int b = 512;
  printf_2("%d %u %hhd\n", b, b, b);

  double c = 2.0;
  printf_3("%d %f %u %hhd\n", c, c, c, c);

  return 0;
}
```

Because the transformed function calls fit such an obvious pattern, the transformation can be easily reversed after running the transformed code through the compiler and catching any problems that it finds. Python scripts that implement this type-safe `printf()` transformer (easily runnable with Python for Windows) are available at [8].

This technique obviously has broader applications beyond aiding in Unicode migrations.

Other issues

These are merely the issues and techniques that I've found most relevant in my own Unicode migrations. Different applications will have different needs.

For example, third-party libraries may need to be upgraded or customized to support Unicode and C++Builder 2009 or 2010. (The most popular libraries, such as JCL, JVCL, DevExpress, and TMS, long ago released Unicode-capable versions.)

Delphi applications that use `String`, `Char`, or `PChar` as byte buffers are heavily affected by the change in size of Delphi's `String`, `Char`, and `PChar` types. C and C++ applications usually use `char` and `char` arrays for byte buffers and so are unaffected, but C++Builder developers who also work in Delphi or who work with heavily Delphi-influenced code may need to be aware of this.

Reading and writing external data now requires attention both to in-memory storage (`RawByteString`, `AnsiString`, or `UTF8String`) and to encodings. (Using the system encoding default ANSI encoding may lose data when transferring from UTF-16. UTF-8 is often preferable.) Code that writes external data also needs to consider writing a Byte Order Mark (BOM), a special sequence of bytes at the beginning of a file that indicates the file's endianness and Unicode encoding.

Finally, database tools and database interactions may require additional attention, depending on your database's capabilities.

Some of these issues are discussed in more detail in [9].

Putting it all together

Unicode is a very broad topic, and even the sub-topic of migrating to Unicode for C++Builder 2009 and 2010 touches upon many techniques. As a review, here's an overview of one approach to migrating your application to C++Builder 2009 or 2010:

First, decide on whether you're going to do a complete migration (to gain the full benefits of Unicode) or a minimal migration (to get up and running in the new IDE as soon as possible).

If you're doing a complete migration:

1. Check your third-party libraries and make sure that they're compatible with C++Builder 2009 and 2010.
2. Before switching to C++Builder 2009 or 2010:
   a. Replace `AnsiString` with `String`.
   b. Mark string literals (`"Hello"`) with tchar.h's `_T` macro.
   c. Replace C library routines with their tchar.h equivalents.
3. Add C++ `typedefs` such as `tstring` so that C++ string manipulation will work after the switch to Unicode.
4. Convert your project to C++Builder 2009 or 2010. Under Project, Options, Directories and Conditionals, make sure that "_TCHAR maps to" is set to "wchar_t."
5. Introduce `AnsiString`, `RawByteString`, and `UTF8String` in places where you need to continue to manipulate narrow character text.
6. Use string shims, C++ overloading, and similar techniques as needed to handle remaining ANSI versus Unicode issues.
7. Run the type-safe `printf()` transformer on your code to catch any issues with variadic macros.
8. Review your code for places where you assume that strings can be arbitrarily indexed or split; this is no longer the case with Unicode.

If you're doing a minimal migration:

1. Check your third-party libraries and make sure that they're compatible with C++Builder 2009 and 2010.
2. Convert your project to C++Builder 2009 or 2010. Under Project, Options, Directories and Conditionals, make sure that "_TCHAR maps to" is set to "char."
3. Replace `String` with `AnsiString`.
4. Use string shims, C++ overloading, and similar techniques to handle interactions between Unicode VCL code and your ANSI application code.
5. Run the type-safe `printf()` transformer on your code to catch any issues with variadic macros.

Gradually switch to Unicode, as time and business cases permit, to gain the full benefits of Unicode.

Contact Josh at **joshkel@gmail.com**.

References

1. Matthew Wilson, *Imperfect C++*. Addison-Wesley, 2004.
2. Matthew Wilson et. al. "STLSoft – Robust, Lightweight, Cross-platform, Template Software." http://www.stlsoft.org/.
3. WideCharToMultiByte. http://msdn.microsoft.com/en-us/library/dd374130%28VS.85%29.aspx.
4. MultiByteToWideChar. http://msdn.microsoft.com/en-us/library/dd319072%28VS.85%29.aspx.

5. "Nick Hodges > Blog Archive > We Won't Overwrite Your Changes." http://blogs.embarcadero.com/nickhodges/2009/05/29/39245.
6. Matthew Wilson. "Generalized String Manipulation: Access Shims and Type Tunneling." http://www.drdobbs.com/cpp/184401689. Retrieved 4/12/2010. (Several of the links within this article are broken; switch to the print view for the complete article.)
7. Nick Galgreath. "Type safe printf." http://blog.client9.com/2008/10/type-safe-printf.html. Retrieved 4/13/2010.
8. Nick Galbreath. "typesafeprintf: type safe printf transformation." http://code.google.com/p/typesafeprintf/. Retrieved 4/13/2010.
9. Cary Jensen. "Delphi Unicode Migration for Mere Mortals: Stories and Advice from the Front Lines." http://edn.embarcadero.com/article/40307. Retrieved 4/13/2010.

Changing C++Builder 2010's Default Save Directory

By Curtis Krauskopf

Versions: C++Builder 2010

The first time I save a project, the C++Builder 2010 (CB2010) IDE always tries to put it in "My Documents\RAD Studio\Projects," which I never use.

In this article, I'll present three solutions which can be used to solve this problem, depending on how badly you want to override the default.

Create a small project

Here's a quick example so you can follow along in your own IDE.

1. Launch CB2010 and add a static library to the default ProjectGroup1. It doesn't matter what you add, but for this example, we'll use a static library. Choose "File | New | Other."
2. In the "New Items" dialog box, choose "C++Builder Projects" from the left-hand selection tree (**Figure 1**) and then select "Static Library" from the right-hand panel.

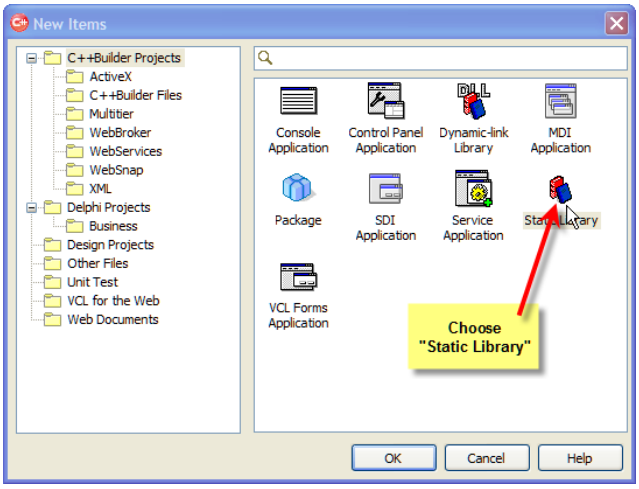
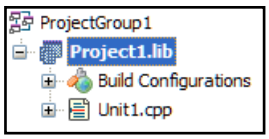


Figure 1: *Create a project using a static library.*

3. In the Project Manager panel, right-click Project1.lib...

and choose "Add New | Unit." You should now see something similar to the following...



4. Right-click on ProjectGroup1 and choose "Save Project Group." A "Save Unit 1 As" dialog box will appear. Its default folder will be "Desktop/My Documents/RAD Studio/Projects."
5. Either left-click on the File name box or use Alt+N to use the shortcut.
6. Enter the name of the drive and folder you want to use.
7. While navigating, you can use the "Create New Folder" icon to create a destination folder...

8. Once you've found the location you want to save the library's CPP file, name the file Library.cpp and then click on the dialog's Save button.
9. A "Save Project1 As" dialog box appears. Save the project into the same folder.
10. A "Save ProjectGroup1 As" dialog box appears. Save the project group into the same folder.

First tip

So far, all of the steps have been the same as when you are saving a regular project. The first tip is that you can use the "My Recent Document" shortcut on the save panel to quickly go back to the folder you previously chose. To see it in action, do these steps:

1. In Project Manager, right-click on ProjectGroup1 and choose "Add New Project."
2. Add another static library to the project.
3. Right-click on the new Project1.lib and choose "Save."
4. The "Save Project1 As" dialog defaults to the "Desktop/My Documents/RAD Studio/Projects" folder. Click on the "My Recent Documents" icon on the shortcuts tray (see **Figure 2**). A list of the folders and documents you've recently worked on will appear. The list defaults to being sorted by chronological order, but you want the most recent folders first.
5. Click on the "Date Modified" title to re-sort the list in reverse chronological order.
6. The folder you want should be at or near the top of the list.

Second tip

Another way to quickly navigate to your work folder is to put a shortcut to the work folder in the "Desktop/My Documents/RAD Studio/Projects" folder.

1. When saving a project, right-click anywhere in the white space in the main panel of the "Save Project1 As" dialog box. A context menu appears.
2. Choose "New | Shortcut" (see **Figure 3**). The Create Shortcut wizard appears.
3. Follow the panels in the Create Shortcut wizard to make a shortcut to your project folder.

Third tip

CB2010 has a way to permanently change the default folder that is used for saving projects.

1. In the CB2010 IDE, choose "Tools | Options." The "Options" panel appears.

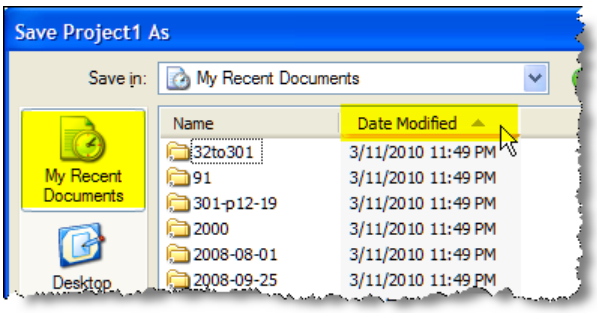


Figure 2: *Create a project using a static library.*

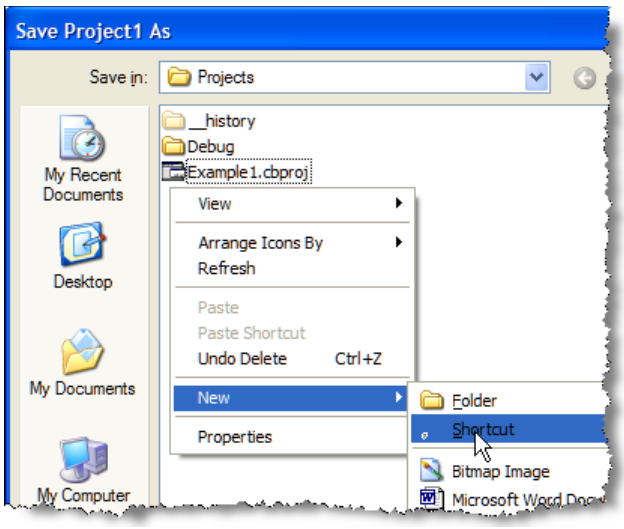


Figure 3: *Make a shortcut in the default project folder.*

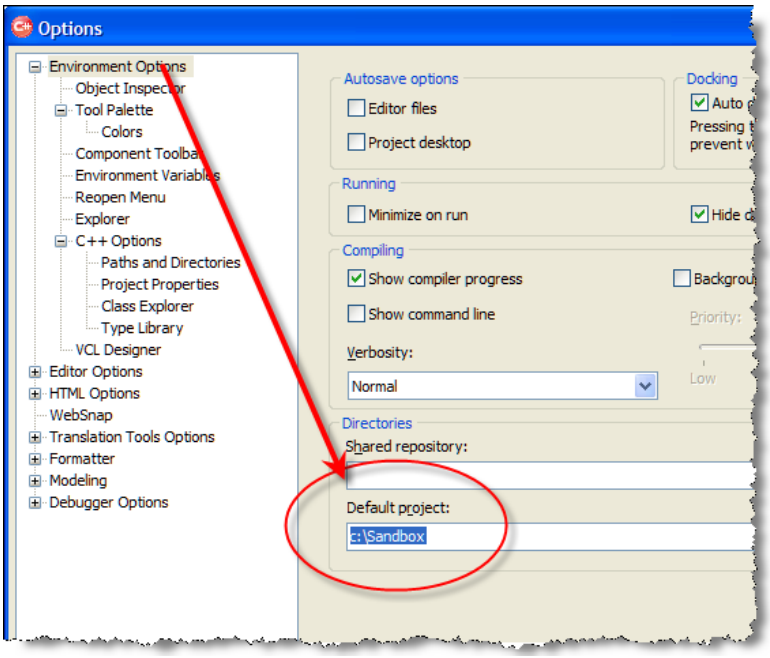


Figure 4: *Define the default project folder within the CB2010 IDE.*

2. The "Options" panel defaults to whatever panel was previously chosen. On the left-side of the panel, navigate to the "Environment Options." It is the first branch at the top of the options tree.
3. At the bottom of the "Environment Options "panel is a "Default project" data entry field (see **Figure 4**). Either enter the name of the project folder or select it using the folder picker (...) to the right of the "Default project" field.
4. Click on the Ok button at the bottom of the form to save your change.
5. All future saves will default to the folder specified in the "Default project" data entry field.

Conclusion

There are three ways to easily navigate to your project folder. I have presented three ways so that you can choose whichever technique (or combination of techniques) best fits your work style and your work environment.

To recap, the techniques are to: (1) Use the My Recent Document icon in the shortcut tray; or (2) Create a shortcut to your work folder; or (3) Specify the default work folder in the CB2010 IDE.

Contact Curtis at **curtis@decompile.com**.

Building a Custom Multi-Touch System

By Byeongcheol Nam and Ki-Tae Bae

Versions: C++Builder 2010

Multi-touch refers to the ability of a touch device to detect and distinguish three of more *simultaneous* touch events. Multi-touch systems have the potential to provide a highly intuitive and productive means of user input (e.g., the iPod).

Multi-touch software programming is a future trend. Through a variety of input devices, many future applications will be required to support multi-touch-based interfaces.

In this article, we describe a custom multi-touch system which uses RS-232 to communicate between a multi-touch device and a C++Builder program. In turn, the C++Builder program (which we'll create) communicates with any multi-touch-enabled application by using a standardized touch protocol (TUIO).

Figure 1: *Custom multi-touch system overview.*

Although Windows 7 has touch support, and although there are a few multi-touch applications out there, most people do not have a multi-touch device. Luckily, however, we can develop our own multi-touch software by using a system-level user-input layer called Multi-Touch Vista [1], which allows you to use two mice to simulate multiple touches. Multi-Touch Vista generates a windows touch message and supports multi-touch programs, such as Microsoft Paint.

This article is recommended for people:

- Interested in multi-touch.
- Who want to use multi-touch software although they don't have a multi-touch device.
- Who want to build their own multi-touch system.
- Who want to make art using multi-touch.

Overview

Figure 1 shows a diagram of the system that we'll create. The multi-touch device communicates with our C++Builder program (the multi-touch signal analyzer) over RS-232. The C++Builder program then communicates with Multi-Touch Vista using the TUIO protocol (via a software library called reacTIVision) [2].

For the purposes of this article, you can use Multi-Touch Vista with two mice that can simulate multi-touch. The following section describes how to install and run Multi-Touch Vista.

Multi-Touch Vista

Multi-Touch Vista is a user input management layer that handles input from various devices and normalizes it against the scale and rotation of the target window [1].

First download Multi-Touch Vista from [1]. You can experience multi-touch to complete this setup process and you need to prepare two mice too. Let's have fun along the journey.

Install Multi-Touch Vista

Multi-Touch Vista is a beta version, but there is no problem using it. After you download Multi-Touch Vista, unzip it.

Open a Command Window: Next, open a command window to install the Multi-Touch Vista driver for your OS. (I will assume your OS is Windows 7 32-bits version.) To do this, open Explorer and look at the inside of the folder to which you extracted the Multi-Touch Vista zip file. Do Shift + Right Click on the "x32" folder in the "Driver" folder to show a popup menu; then select "Open Command Window" (see **Figure 2**).

Install the Driver: Run "Install driver.cmd" from the command line. A Confirm Message Window will arise; click "Yes" to finish installing the driver. If you are running a 64-bit version of Windows, then run a command-line as an administrator.

Reload the Device: Close the command window and go to Control Panel and then select Device Manager. Expand the Human Interface Devices. Reload the Universal Software HID device. It is created when you running "Install driver.cmd" (see **Figure 3**). Right-click on "Universal Software HID device," then select "Disable," and then answer "Yes" for the prompt. Finally, select "Enable" to reload the device.

Confirm the Installation: To confirm the installation, go to Control Panel and then System to check that "Pen and Touch" indicates "Touch Input Available with 255 Touch Points" as shown in **Figure 4**.

Run Multi-Touch Vista

There are two ways to run Multi-Touch Vista. One is to run it in console mode. The other is to register and run it as a service.

Console mode

Let's look at the downloaded folder. In there, you should see the following files:

(1) Multitouch.Service.Console.exe
(2) Multitouch.Driver.Console.exe
(3) Multitouch.Configuration.WPF.exe

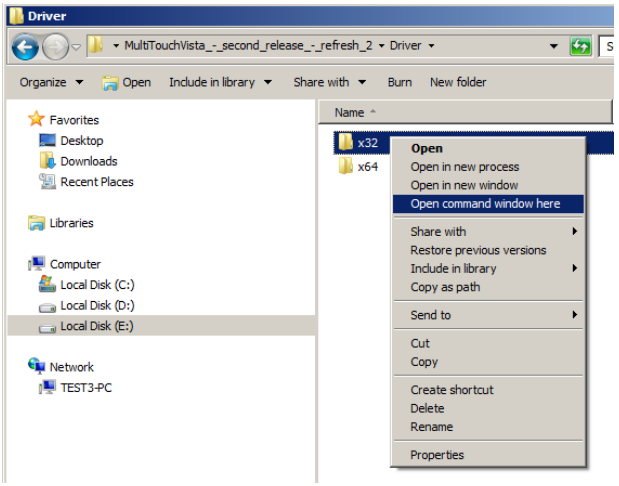


Figure 2: *Open a command window in the unzip folder.*

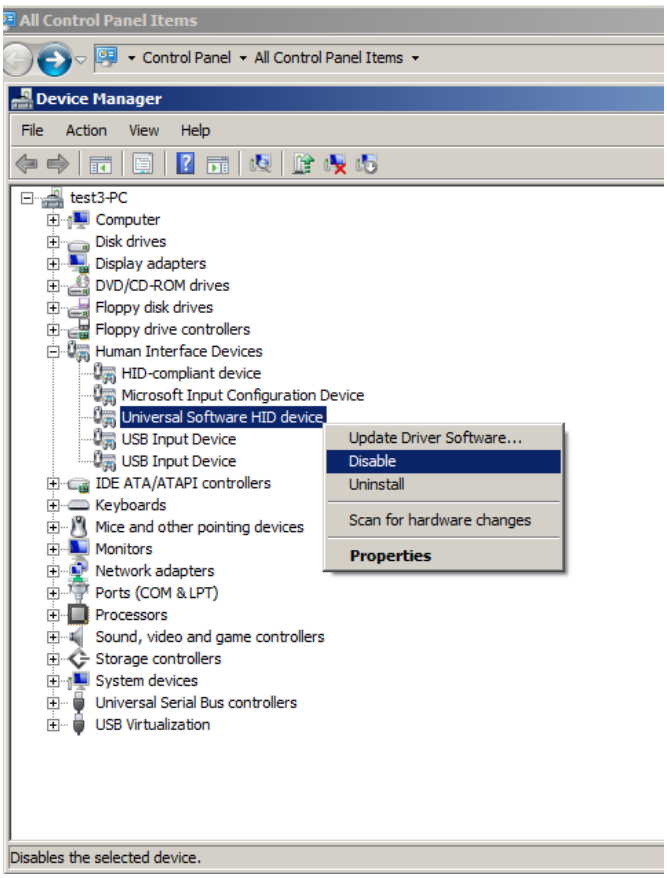


Figure 3: *Several Universal Software HID devices are installed.*

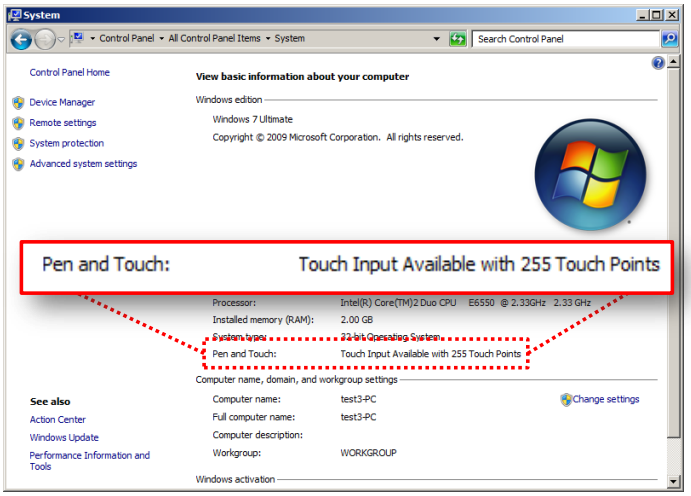


Figure 4: *Multi-Touch Vista drivers installed.*

Run (1) and (2) sequentially to use MultipleMice or TUIO. As shown in **Figure 5**, to switch between MultipleMice and TUIO run (3). The default input is MultipleMice so you can see red dot(s), but you still have to use the regular mouse cursor to interact with Windows. Therefore you should run (3) then select "Configure Device" to mark the check box. When you do this, the red dot can interact with Windows.

To test the setup, you can run Microsoft Paint or the PickStack program as shown in **Figure 6**.

Service mode

If the console mode test is gracefully over then you want to know how to setup service mode. With service mode, there is no need to run many executable files. (However, I recommend you to use console mode while you are developing something.)

First, launch a command window using the "Run as administrator" option (see **Figure 7**).

At the command window, navigate to the Multi-Touch Vista folder, and then register the service by typing:

```
C:\Windows\Microsoft.NET\Framework\v2.0.50727\
installutil.exe /i Multitouch.Driver.Service.exe
```

You should see text similar to that shown in **Figure 8**.

The default Multi-Touch service is set to manual mode. You will likely want it to start automatically, so you should change the service into "auto" mode. To do this, from the Start menu, select "Administrative Tools | Services." Right-click the entry for "Multi-

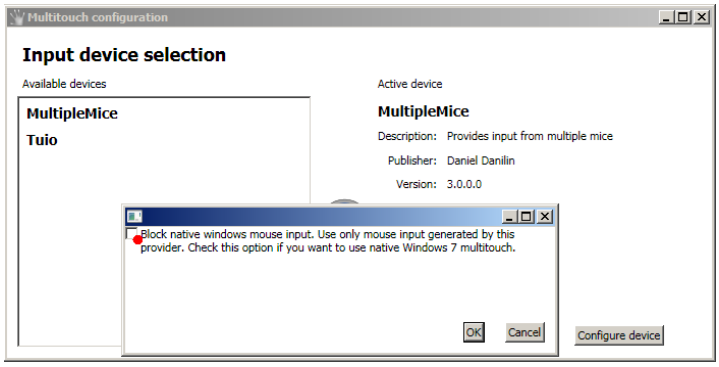


Figure 5: *Multi-Touch Vista drivers installed.*

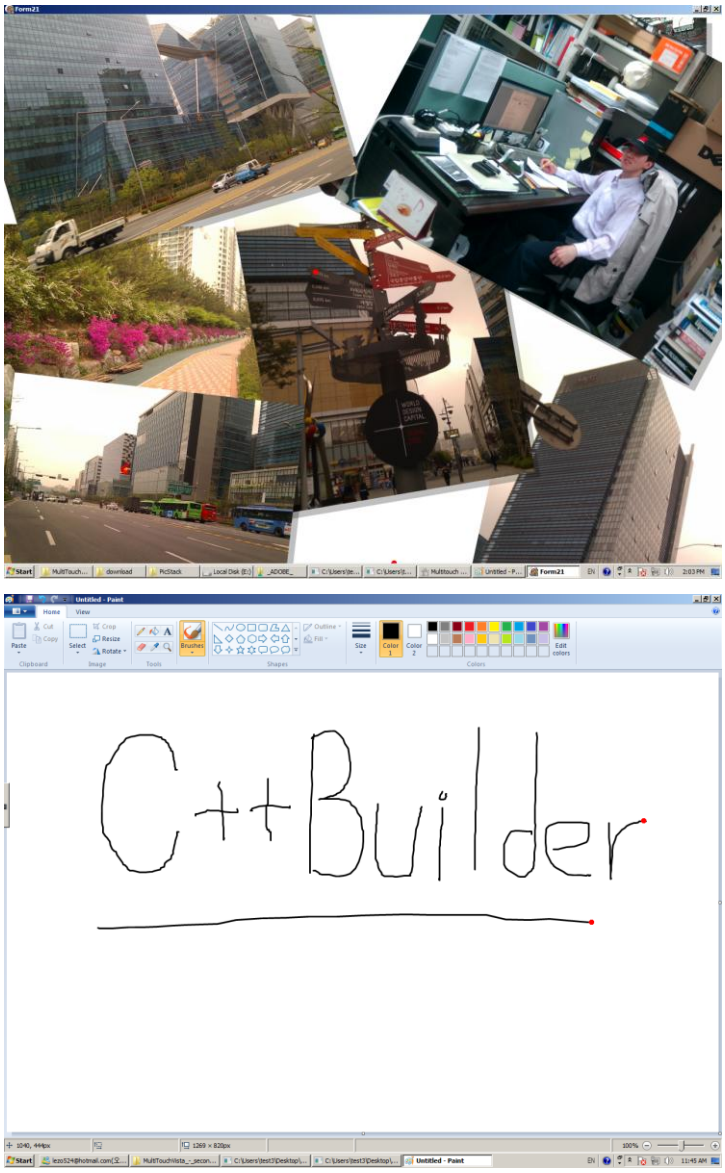


Figure 6: *Multi-Touch-enabled applications. Top: PickStack.exe (written in Delphi). Bottom: MS Paint.*

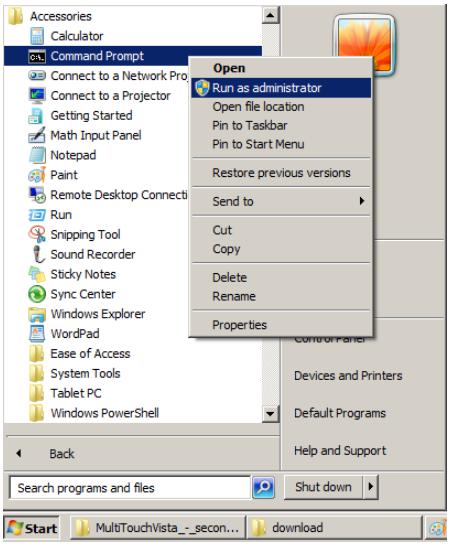


Figure 7: *Running a command prompt as an administrator.*

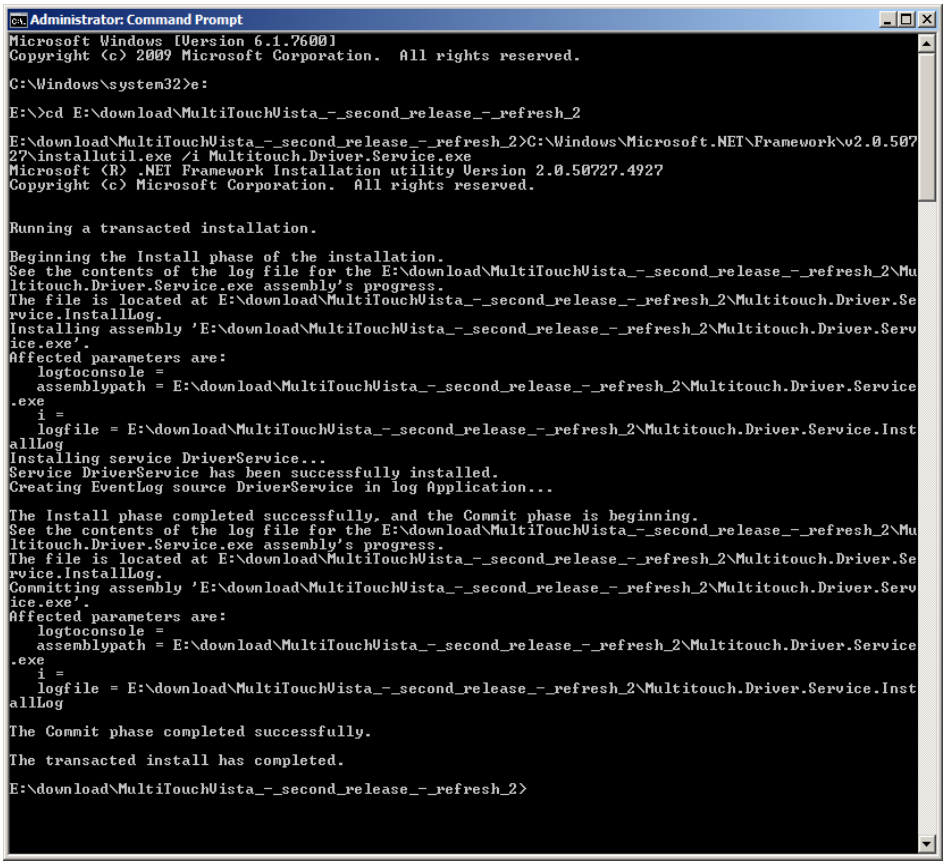


Figure 8: *Starting the service.*

touch Driver," select "Properties," and then change the "Startup type" field to "Automatic."

To unregister the service, you can launch the command:

```
C:\Windows\Microsoft.NET\Framework\
v2.0.50727\installutil.exe /u
Multitouch.Driver.Service.exe
```

At this point, installation of the core modules of Multi-Touch Vista has been completed. Next, we'll look at using the CPort component to receive RS-232 data. Here, we'll use the CPort 4.0 Beta for C++Builder 2010.

CPort 4.0 Beta

The ComPort (CPort) library is a set of Delphi/C++Builder components for serial communication. It's an open-source project available at [3].

Unfortunately, there is no CPort 4.0 project file for C++ Builder 2010, but you can use the project file that I've created and included with the code that accompanies this article. The component project file name is "TComPort Delphi 2010", so you could use it with C++ Builder 2010. Build "CPortLibCB2009.bpl" and then "DsgnCPortCB2009.bpl". If you build them all, you should install "DsgnCPortCB2009.bpl" as shown in **Figure 9**.

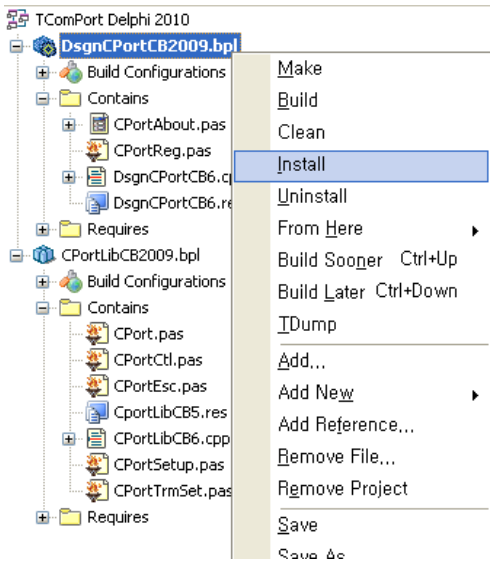


Figure 9: *Installing the ComPort library.*

Upon installation, you should see the CPortLib tab in the Component Palette (**Figure 10**).

Now, drop a `TComPort` component (the first component on the tab) onto a form. To use the component, do the following steps. The code shown below are snippets from the demo application that accompanies this article. `TTouchForm` is the demo's main form.

1. *Set up the COM port*: Open the setup dialog to initialize the RS-232 component; the dialog can be shown, e.g., in

response to a button click:

```
void __fastcall TTouchForm::
  SETUPRS2321Click(TObject *Sender)
{
  ComPort->ShowSetupDialog();
}
```

This will result in the setup dialog being shown (see **Figure 11**).

2. *Open the COM port*: If you have finished RS-232 initialization then you can begin to communicate with other devices by first opening the COM port:

```
void __fastcall TTouchForm::
  ComPortOpen1Click(TObject *Sender)
{
  if (ComPort->Connected)
    ComPort->Close();
  else
    ComPort->Open();
}
```

3. *Receive RS-232 data*: We will receive data from a touch device only, so we need to code only the receiver side (`OnRxChar` event handler):

```
void __fastcall TTouchForm::
  ComPortRxChar(TObject *Sender, int Count)
{
  AnsiString asStr;
  ComPort->ReadStr(asStr, Count);

  // Parsing Code Here
}
```

Now we are ready to receive RS-232 data from the touch device. The next step is to use SimpleSimulator, written in C++ using the TUIO Protocol, to send the data to Multi-Touch Vista.

Using TUIO

TUIO is an open framework that defines a common protocol and API for tangible multi-touch surfaces. Therefore TUIO is an ideal choice for making a custom Multi-Touch system.

In this article, we will use reacTIVision's SimpleSimulator C++ class, which uses the TUIO protocol to send a message to Multi-Touch Vista. You can get the SimpleSimulator source code at [4]. I have modified the code to make it easier to use for C++Builder developers; these modifications are included with the

Figure 10: *The CPort components.*

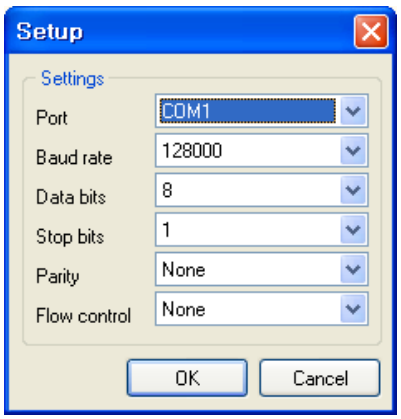


Figure 11: *The COM port setup dialog.*

code that accompanies this article.

For reference, the declaration of the `SimpleSimulator` class is as follows:

```
using namespace TUIO;

class SimpleSimulator {

public:
  SimpleSimulator(const char *host,
    int port, int cursor_range);
  ~SimpleSimulator() { delete tuioServer; };

  TuioServer *tuioServer;
  std::vector<TuioCursor*> m_cursor_vector;
  void run();
  void preProcess();
  void postProcess();
  void mousePressed(int id, float x,
    float y);
  void mouseReleased(int id, float x,
    float y);
  void mouseDragged(int id, float x,
    float y);

private:
  void toggleFullscreen();

  // other members not shown (see code)
};
```

There are five important functions in the class: `preProcess()`, `postProcess()`, `mousePressed()`, `mouseReleased()`, and `mouseDragged()`. You can generate a touch message using the latter three functions

placed between calls to `preProcess()` and `postProcess()`. The three functions are declared as follows:

```
mousePressed( int id, float x, float y );
mouseReleased( int id, float x, float y );
mouseDragged( int id, float x, float y );
```

where `ID` represents each (`x`, `y`) touch point. If ID is different from others, that indicates a different touch point. Note that `x` and `y` are relative coordinate.

Another advantage of using TUIO protocol is that it is based on UDP communication. So it could be used between remote PCs at the same time.

Touch signal analysis program

Finally, let's take a brief look at the Touch Signal Analysis Program (this article's demo program). Here's a condensed version of the main form's header (`TTouchForm`):

```
class TTouchForm : public TForm
{
__published:  // IDE-managed Components
  TComPort *ComPort;
  void __fastcall SETUPRS2321Click(
    TObject *Sender);
  void __fastcall ComPortRxChar(
    TObject *Sender, int Count);

public:     // User declarations
  __fastcall TTouchForm(TComponent* Owner);
  __fastcall ~TTouchForm();

  ZGridView* m_pGrid;
  TCanvas* m_pTouchCanvas;
  SimpleSimulator* m_pTouch;
  ZTouchPoint* m_pTouchPoint;

public: // RS232
  AString m_aReceived;
  AnsiString m_asReceived;
  void __fastcall UpdateRS232(int Count);
};
```

The program receives RS-232 data from the touch device via the `TComPort` component. When data arrives, the `TComPort` component's `OnRxChar` event handler calls the following function (the main update function):

```
void __fastcall TTouchForm::
  UpdateRS232(int Count)
{
  AnsiString asRead;
  AString aDummyString, aToken;
  int iST = 0, iEND = 0;

  // Receive RS-232 Data
  ComPort->ReadStr(asRead, Count);

  m_aReceived += asRead.c_str();
  m_aReceived.TrimLeft("END");
  iST = m_aReceived.Pos("ST");
  iEND = m_aReceived.Pos("END");

  // Parse RS-232 Data
  // ST[1-00,00,00,00,00,00,00,00]
  // [21-00,00,00,00,00,00,00,00]END
  if (iST >= 0 && iEND > 0 && iST < iEND) {

    // ST ~ END means a single frame.
    m_aReceived.TrimLeft("ST");
    aDummyString =
      m_aReceived.Token("END");

    //------------------------------
    m_pTouchPoint->ResetGridPointList();
    m_pTouchPoint->ResetTouchList();
    do
    {
      // [ ~ ] means a single column line
      aDummyString.TrimLeft("[");
      aToken = aDummyString.Token("]");

      if (!aToken.IsEmpty()) {
        // Analysis and Add every sensing
        // point.
        m_pGrid->Analysis( aToken );
      }

    } while(!aToken.IsEmpty());
    //------------------------------

    // Grouping points.
    m_pTouchPoint->MakeGroup();
  }

  //------------------------------
  // UPDATE & RENDER
  //------------------------------
  if( Prm.enable_touch_message )
    m_pTouch->preProcess();

  if( Prm.update_touch_point )
    m_pTouchPoint->Update(
      Prm.touch_point_update_time );

  if( Prm.render_touch_point )
  {
    m_pTouchPoint->RenderPoint();
    m_pTouchPoint->RenderGroup();
    m_pTouchPoint->RenderTempTouch();
    m_pTouchPoint->RenderSelectedTouch();
  }

  if( Prm.enable_touch_message )
    m_pTouch->postProcess();
  //------------------------------
}
```

The brains of this code lies in the grouping algorithm, which is handled by another class `ZTouchPoint`. (The `m_pTouchPoint` variable in the above code is of type `ZTouchPoint`.) This latter class performs the steps shown in **Figure 12**.

The touch device just consists of sensing points. As shown in **Figure 12**, `ZTouchPoint::MakeGroup()`, changes these sensing points into grouping points. Now, we need to know for sure where the touched point is. So, using `CalcMeanPoint()`, the class calculates the center point.

Next, we need to decide what touch point is real; this is the role of the `SelectActualPoint()` method. Properly determining whether an actual point was touched is one of most important things that deteremines the credibility of a multi-touch system with an infrared sensing device. We recommend you to make your own algorithm—tuned to your touch device—and place it within the `SelectActualPoint()` method.

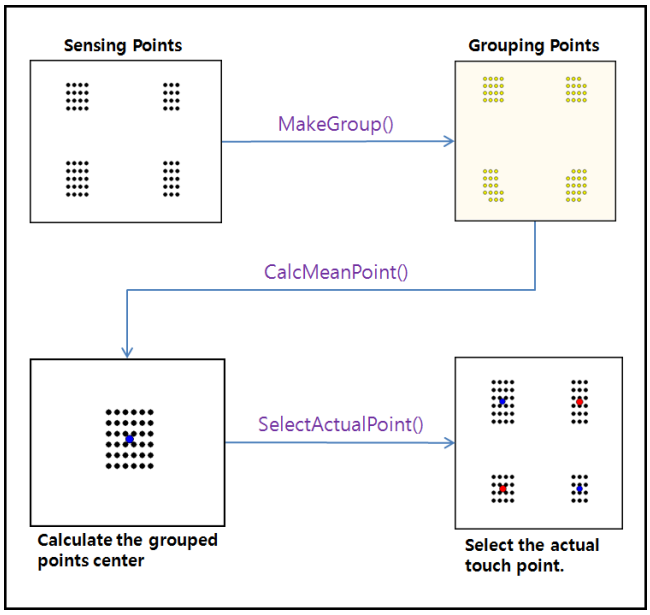


Figure 12: *The grouping algorithm.*

Conclusions

We have been looking for a low-cost multi-touch system. Using an infrared sensor device has known to be suitable for a single touch. But we need to have a multi-touch system for cheap and efficient development.

Here, I've demonstrated how to use Multi-Touch Vista and the `TComPort` component to accomplish this task. A key advantage of the approach presented here is that it has creative possibilities for use over a network. This article and its accompanying source code (particularly, the demo application) should provide you a good start to creating your own multi-touch system. Now, the real journey begins... Enjoy!

Contact ByeongCheol at **lezo524@gmail.com**.
Contact KiTae Bae at **ktbae@kgit.ac.kr**.

This research was supported by Seoul R&BD Program (PA090701).

References

Cited

1. Multi-Touch Vista
http://multitouchvista.codeplex.com/
2. TUIO Library
http://www.tuio.org/
3. CPort component
http://sourceforge.net/projects/comport/
4. ReacTIVision 1.4
http://reactivision.sourceforge.net/

Other

5. Install Multi-Touch Vista
http://wenjiun.blogspot.com/2009/11/testing-windows-7-multi-touch-with.html
6. CodeGear Guru
http://codegearguru.com/
7. C++ Builder DOCs
http://docs.embarcadero.com/products/rad_studio/
8. Nick Hodges
http://blogs.embarcadero.com/nickhodges/
9. C++ Builder Lecture
http://www.yevol.com/en/bcb/index.htm
10. C++ Builder IDE & Compiler Boosting
http://andy.jgknet.de/blog/
11. Imp: Delphi/C++Builder Evangelist in Korea
http://blog.devgear.co.kr/
12. Devgear Products Page in Korea
http://www.devgear.co.kr/products/cbuilder/

http://forums.bcbjournal.org

Discuss C++Builder with other developers in our online forum

Visit our forum to **discuss anything C++Builder** with other C++Builder developers. Our forum is open to everyone and provides a direct link to our editors, authors, and the entire C++Builder Developer's Journal community.

This is the perfect place to discuss articles/code, and to get quick answers to your technical and non-technical questions. We also encourage you to browse our archive of monthly poll questions and to participate in forthcoming polls.

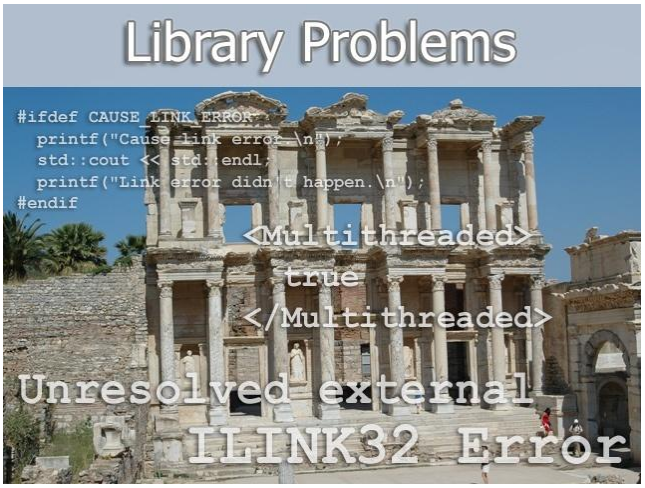
We particularly welcome your feedback on this special issue and suggestions for future special issues!

Library Problems

By Curtis Krauskopf

Versions: C++Builder 2010, 2009

(This article was first published in the April 2010 issue of the C++Builder Developer's Journal, Vol. 14, Num. 4.)



My head hurts. I recently upgraded my development environment from BCB 6 to C++Builder 2010 (CB2010). All was going well and I was becoming comfortable with my new development environment. Therefore, I decided to port some libraries I've been using to CB2010.

That went fine too. There were no new compiler warnings and I thought this whole upgrading process would be simple. The regression tests even passed without any problems.

But, that's when Murphy came knocking at my door [1]. The first sign of problems was when I compiled the main part of a large program that used several newly recompiled libraries. ILINK32 reported that four externals were unresolved (see **Figure 1**).

These link errors are saying that four functions (`__InterlockedDecrement`, `__InterlockedIncrement`, `__Unlocksyslock`, and `__Locksyslock`) are used by Library.lib but they are not defined in any of the libraries or the application's modules. The two leading underscores on the names told me that these were probably symbols in the compiler's library.

Researching the problem

CB2010 help wasn't very helpful because none of these symbols were in the help's index or in the local database. There were some search hits in the MSDN Online, Codezone Community and Questions sections, but they were for Visual Studio.

Searching the CB2010 include files showed they are in the YVALS.H file in the C++ library (~\include\dinkumware\yvals.h). The following code shows the relevant code from YVALS.H.

```
// Code snippet of yvals.h...
/* MULTITHREAD PROPERTIES */
#if _MULTI_THREAD
  _EXTERN_C
    _CRTIMP2 void _Locksyslock(int);
    _CRTIMP2 void _Unlocksyslock(int);
    long _CRTIMP2
      _InterlockedIncrement(long *);
    long _CRTIMP2
      _InterlockedDecrement(long *);
  _END_EXTERN_C
#else /* _MULTI_THREAD */
  #define _Locksyslock(x)   (void)0
  #define _Unlocksyslock(x) (void)0
  #define _InterlockedIncrement(x) \
    (++(*x))
  #define _InterlockedDecrement(x) \
    (--(*x))
#endif /* _MULTI_THREAD */
```

This code declares `_Locksyslock` (and the other functions) as prototypes when `_MULTI_THREAD` is defined and set to a non-zero value. Otherwise, `_Locksyslock` and `_Unlocksyslock` are given null definitions and `_InterlockedIncrement` and `_InterlockedDecrement` are defined to be increment and decrement operations.

The problem seems to be that a library (Library.lib) was compiled with a multithread dependency.

```
[ILINK32 Error] Error: Unresolved external '__InterlockedDecrement' referenced from C:\CPP\DEBUG\LIBRARY.LIB|LibrarySrc
[ILINK32 Error] Error: Unresolved external '__InterlockedIncrement' referenced from C:\CPP\DEBUG\LIBRARY.LIB|LibrarySrc
[ILINK32 Error] Error: Unresolved external '__Unlocksyslock' referenced from C:\CPP\DEBUG\LIBRARY.LIB|LibrarySrc
[ILINK32 Error] Error: Unresolved external '__Locksyslock' referenced from C:\CPP\DEBUG\LIBRARY.LIB|LibrarySrc
```

Figure 1: *Four link-time errors for unresolved symbols. The original library name is unimportant and it has been replaced with 'Library.lib' for this article.*

I had two questions at this point:

1. How did the library get a multi-thread flag set when I didn't specify that the library would be compiled in multi-thread mode?
2. Why did the regression tests pass for the library? The regression tests should have also gotten a link-time error when linking the library.

Default Multi-threaded libraries?

I checked all of the panels in the Project Options for the library. Nothing in there indicated that the library would be compiled in a multi-threaded mode.

The application was single-threaded because I had specified that in the console wizard when I created the application's project (see **Figure 2**).

I even double-checked the static library wizard by creating a test library just to see if there was a multi-threaded setting I missed there. No—there was nothing in the wizard letting me specify a single-threaded or multi-threaded library.

That led me to my next question: how does a project know, at compile-time, if it's supposed to be single-threaded or multi-threaded?

How does a project know?

I created two console projects. In the console wizard, I activated the multi-thread checkbox in one project and deactivated it in the other project.

This time, I knew I had two projects side-by-side that had to have something different about them.

Checking every panel in the Project Options proved fruitless. It wasn't defined anywhere there.

The only place it could be defined, then, would be in the project's .CBPROJ file. I opened both .CBPROJ files in a separate editor (opening it CB2010 cleverly opens the project in the CB2010 IDE). The .CBPROJ files are XML files.

Figure 3 shows the result of using my favorite text comparison tool. It shows that a multi-threaded

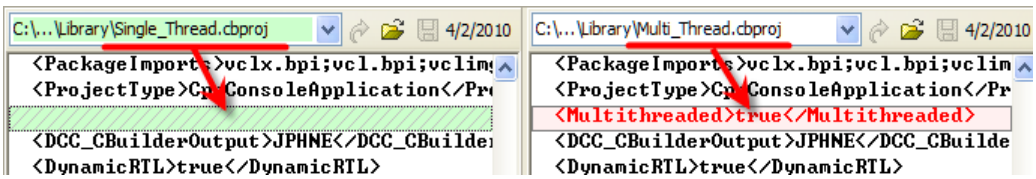


Figure 3: *A multi-threaded application has an XML property called Multithreaded set to true. A single-threaded application does not have the XML property at all.*

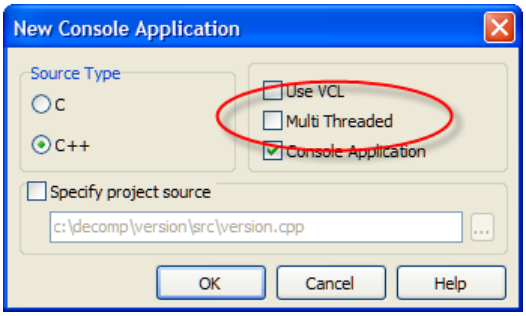


Figure 2: *On the New Console Application wizard, the Multi Threaded checkbox determines if the application will be single-threaded or multi-threaded.*

project has an XML entry called Multithreaded in a PropertyGroup. The PropertyGroup is distinguished from the other PropertyGroups in the XML file by having this condition:

```
Condition="'$(Base)'!=''"
```

I know it looks strange but it's not important to understand it right now.

According to the CB2010 help, "At compile-time, the compiler sets the predefined macro, __MT__, to 1 when the -tWM compiler option is set."

The code I used to test for __MT__ being defined looked like this:

```
#include <stdio.h>

int main(int, char**)
{
  printf("In " __FUNC__ "()\n");
  #ifdef __MT__
    printf("In " __FILE__
           ", __MT__ = %d\n", __MT__);
  #else
    printf("In " __FILE__
           ", __MT__ not defined\n");
  #endif
  return 0;
}
```

If __MT__ is defined at compile-time, the code displays its value; otherwise, the code displays "not defined."

In my test programs, __MT__ was undefined in single-threaded project and it was defined in multi-threaded project. Just to test my theory, I used an

edit feature in my favorite text comparison tool to move the Multithreaded XML line from the multi-threaded project into the single-threaded project.

Success! Recompiling the programs showed that the single-threaded project was now compiling with __MT__ defined and the multi-threaded project was compiling with __MT__ undefined. That's exactly the kind of behavior I expected by moving the Multithreaded XML property from the multi-threaded project into the single-threaded project.

Back to the library

When I opened the library's .CBPROJ file in the text editor, lo and behold, there was a Multithreaded XML line in the PropertyGroup with the same condition as I found in the project's application.

It appears, in CB2010, that static libraries default to being compiled in multi-threaded mode and there is no way to change it in the IDE or even detect it in the Project Options panel.

That answered the first question, but what about the second one? Why did the regression test pass?

The regression test was compiled using a .MAK file, not from a .CBPROJ. Opening the .MAK file explained why there was no compile-time problem with the regression test: the -tWM compiler flag was enabled. The regression test's application was compiled in multi-thread mode.

Exploring undocumented behavior

Static libraries default to being compiled with the multi-threaded mode enabled. If there isn't a way to turn that feature off in the IDE, can I disable it in the .CBPROJ file?

To find that out, I opened the .CBPROJ file using an editor other than CB2010. I found the

```
<Multithreaded>true</Multithreaded>
```

line and changed its property to false.

When I recompiled, I was surprised to find that multi-threading was still enabled in the static library! Using the example I had from the single-threaded application in which the property wasn't even defined, I then deleted the Multithreaded property from the .CBPROJ file. After recompiling, the library was now compiled without multi-threading enabled.

But wait, there's more

If that was all there was to the problem, then this article would be finished. We discovered:

- static libraries default to multi-threaded
- a static library compiled with the _MULTI_THREAD symbol defined causes a link-time error when the library is linked with a single-threaded main module.
- an undocumented XML property called Multithreaded controls whether the IDE compiles the module in multithreaded mode
- CB2010 compiles the module in multi-threaded mode even if the XML Multithreaded property is set to false
- CB2010 can compile static libraries in single-threaded mode when the undocumented Multithreaded property is removed from the .CBPROJ file.

The previous code snippet from YVALS.H used _MULTI_THREAD instead of __MT__. Even though the symbol's name gave us the right clue, it was for the wrong reason. Here's why: The real dependency is when the dynamic runtime library (RTL) is linked into the executable.

The dynamic RTL is specified in the base, debug and release configurations on the C++ Linker | Dynamic RTL property. When the dynamic RTL is true, that signals the linker to link the runtime library calls to an external cc3290.dll file. When the dynamic RTL is false, the RTL is compiled into the executable so that it becomes stand-alone -- you only need to distribute the executable without distributing any Embarcadero (Borland) runtime library files.

It's actually much worse than that. CB2010 and CB2009 will compile a program that causes an access violation at runtime when the library calls std::endl when the RTL is dynamically linked.

Did I mention that my head hurts?

The following code shows an example of what I'm talking about so you can follow along.

```
// Shows how to cause an access violation
// in a perfectly legitimate C++ program

// LibrarySrc.hpp
#ifndef LIBRARYSRC
#define LIBRARYSRC
void call_Library();
#endif
```

```
// Library.cpp:
#include <stdio.h>
#include "LibrarySrc.hpp"
#include <iostream>

// #define CAUSE_LINK_ERROR
// #define CAUSE_AV_ERROR
void call_Library() {
  printf("In %s()\n", __FUNC__);
  #ifdef __MT__
    printf("In " __FILE__
           ", __MT__ = %d\n", __MT__);
  #else
    printf("In " __FILE__
           ", __MT__ not defined\n");
  #endif

  #ifdef CAUSE_LINK_ERROR
  printf("Cause link error.\n");
  std::cout << std::endl;
  printf("Link error didn't happen.\n");
  #endif

  #ifdef CAUSE_AV_ERROR
  printf("Cause runtime AV error.\n");
  std::cout << "AV error.\n";
  printf("AV error didn't happen.\n");
  #endif
}

// Main.cpp
#include <stdio.h>
// #include <vcl>
#include "LibrarySrc.hpp"

int main(int, char**)
{
  printf("In " __FUNC__ "()\n");
  #ifdef __MT__
    printf("In " __FILE__
           ", __MT__ = %d\n", __MT__);
  #else
    printf("In " __FILE__
           ", __MT__ not defined\n");
  #endif
  call_Library();
  return 0;
}
```

The library.cpp in the above code has two compile-time symbols commented out: `CAUSE_LINK_ERROR` and `CAUSE_AV_ERROR`.

When `CAUSE_LINK_ERROR` is uncommented and the main module is compiled with a dynamic RTL, the compiler will complain that four symbols are not defined at compile-time (see **Figure 1**).

When `CAUSE_AV_ERROR` is uncommented (and `CAUSE_LINK_ERROR` is commented) and the main module is compiled with a dynamic RTL, the runtime will cause an access violation when it executes the `std::cout`.

ST Console Mode						
std::endl in lib	N	N	Y	N	Y	N
std::cout in lib	N	N	Y	Y	Y	Y
Dynamic RTL	N	Y	N	N	Y	Y
Result:						
Compiles?	Y	Y	Y	Y	N	Y
AV ERROR @ RUNTIME?	N	N	N	N	-	Y
MT Console Mode or VCL Console Mode						
std::endl in lib	N	N	Y	N	Y	N
std::cout in lib	N	N	Y	Y	Y	Y
Dynamic RTL	N	Y	N	N	Y	Y
Result:						
Compiles?	Y	Y	Y	Y	Y	Y
AV ERROR @ RUNTIME?	N	N	N	N	Y	Y

Table 1: *Lists when link-time errors and run-time errors occur for the sample program.*

What is the most insidious about this runtime error is that your regression tests might not ever encounter this problem if the library uses `std::cout` or `std::cerr` for error messages and the regression tests never exercise those lines. This problem could lie dormant inside the executable for years and then the one time a message is emitted by the library, instead of getting an error message, you'll get an access violation from the operating system.

Digging into this problem to bracket it, I found that AV errors occur for the code in Listing B depending on these situations:

- If the project is compiled for single-thread, multi-thread or VCL
- If the library calls `std::endl`
- If the library calls `std::cout` or `std:cerr` at runtime
- If the dynamic RTL is enabled

All 18 permutations are enumerated in **Table 1**.

Table 1 shows that the analysis for the first part of this article was wrong: the problem isn't due to whether or not the library and main console are compiled for single-threaded or multi-threaded, the real cause of the problem is when the dynamic RTL is linked into the project.

So why did the original regression test really pass? It's because the regression test's .mak file was linking the runtime library into the executable statical-

ly (dynamic RTL was disabled). However, all new CB2010 console-mode projects default to having the dynamic RTL linked with the project.

What to do about this?

Based on the permutations shown in **Table 1**, if your project links to a static library that *might* have `std::cout`, `std::err`, or `std::endl`, you can not also link to a dynamic RTL. This is easy enough to change by modifying the base build configuration for the project but you'll also need to verify that the debug and release builds do not override your new default.

Conclusion

CB2009 and CB2010 have a significant potential problem when linking static libraries into a project. If the static library calls `std::endl` and dynamic RTL is enabled (by default) in the project, you will get a compile-time error for four undefined symbols (**Figure 1**).

However, if the static library doesn't use `std::endl` but it does use `std::cout` or `std::err`, you won't get a compile-time error but the application will throw an access violation the first time it tries to execute that code.

The solution is simple: whenever you link static libraries, always disable the dynamic RTL in the project's options. Because a `#pragma` can link in static libraries without you specifying it in the project's options, the safest solution is to always disable the dynamic RTL in all of your projects.

As a side-effect of the investigation, this article also shows how to determine if a module is compiled for single-thread or multi-thread operation. An undocumented feature in CB2010 is that libraries default to being compiled in multi-threaded mode. By removing the Multithread XML property from the .CBPROJ file, a library can be compiled in single-thread mode.

Contact Curtis at **curtis@decompile.com**.

Acknowledgement

I want to thank Malcolm Smith for verifying that this problem exists in CB2009.

References

1. http://en.wikipedia.org/wiki/Murphy's_law

This Month's Developer's Poll

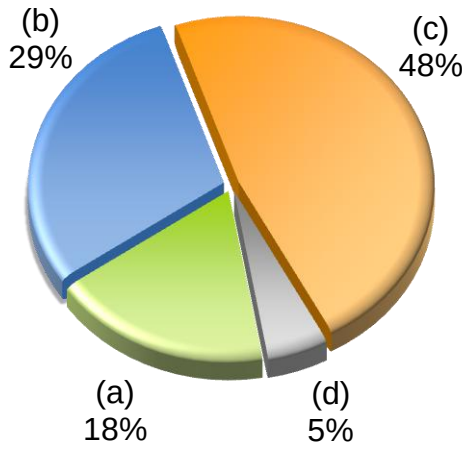
Each month, the Journal's Developer's Poll allows you to compare your opinions to those of other members of the C++Builder Developer's Journal community. The results of this month's poll will be published in next month's issue.

Last month's poll question was:

How often do you upgrade C++Builder?

a. Every new release
b. Every other release
c. Only on occasion
d. Never

The results are shown in the following chart:



These results are based on 38 total votes.

This month's poll question is:

If the next C++Builder supports cross-platform development, will you upgrade?

a. Definitely, yes!
b. Maybe
c. No, cross-platform development is a non-issue for me

Cast your vote online at
http://polls.bcbjournal.org

This Month's Contributors

Malcolm Smith msmith@bcbjournal.org

Editor-at-Large **Malcolm Smith** is owner of MJ Freelancing, which develops custom components and bespoke projects. Malcolm is also a contributing author of the C++Builder 5 Developer's Guide and he is Chief Analyst Programmer for Comvision Pty Ltd. designing and implementing security management systems, concentrating on the integration of disparate CCTV and alarm systems as well as streaming digital video into security control rooms.

Bob Swart Bob@eBob42.com

Contributing Editor **Bob Swart** (aka Dr.Bob) of Bob Swart Training & Consultancy (www.drbob42.com) is based in The Netherlands, and offers Delphi training and private workshops, courseware manuals (for self-study), consultancy and development, and is an Advantage and Embarcadero reseller for the BeNeLux and UK.

Josh Kelley joshkel@gmail.com

Josh Kelley is a lifelong software developer and a husband, father of three, Christian, cyclist, and gamer. He works as a cross-platform C++ developer for Windrock, Inc., a designer of diagnostic systems for industrial machinery.

Curtis Krasukopf curtis@decompile.com

Contributing Editor **Curtis Krauskopf** is a systems engineer and owner of The Database Managers (www.decompile.com). He has been writing code professionally for over 25 years. His prior projects include flight simulator and serious game technologies, decompilers for the DataFlex language, and multiple web e-commerce applications. Curtis has spoken at many domestic and international DataFlex developer conferences and has been published in FlexLines Online, JavaPro Magazine, C/C++ Users Journal, and C++Builder Developer's Journal. Curtis is available for both short and long-term projects.

ByeongCheol Nam lezo524@gmail.com
Ki-Tae Bae ktbae@kgit.ac.kr

Byeongcheol Nam received the B.S. degree in Computer Engineering from Dongeui University, Korea, in 1998 and 2005, respectively. He is studying master's course in the Intelligent Media Lab at the Korean German Institute of Technology. His research interest is a game development (AI and servers). He has worked for 7 years as a professional game developer in MMORPG Server and Online Game AI. The last product that he worked on was ROSE Online by Triggersoft, published by GRAVITY Co. in 2006.

Ki-Tae Bae received the M.S. degree in Computer Engineering and the Ph.D degree in Computer Information Engineering from Chonnam National University, Korea, in 1999 and 2006, respectively. He is an associate professor in the Department of New Media at the Korean German Institute of Technology. His research interests include computer vision, intelligent agents and media processing, and various topics in software engineering.

C++BUILDER Developer's Journal
EnCoded Communications Group

C++Builder Developer's Journal (ISSN 1093-2097) is published online monthly by Encoded Communications Group, 5608 E. 68th St., Stillwater, OK 74074 USA.

Customer Service: support@bcbjournal.com
Customer Relations (Voice) (405) 385-9190
Customer Relations (Fax) (707) 238-3031

Send all written correspondence to:
EnCoded Communications Group
5608 E. 68th St.
Stillwater, OK 74074 USA

Editorial: editor@bcbjournal.org
Editor-in-Chief, Copy Editor Damon Chandler
Editors-at-Large Malcolm Smith
George Tokas
Contributing Editors Bob Swart
Mike Collins
Curtis Krauskopf

Price

Personal	$49/year
Personal with email PDF delivery	$52/year
Corporate/Library/Institutional	$79/year