CAVE MAN'S GUIDE TO CONVERTING DELPHI CODE TO PRISM

rev-02

this is a mostly-finished work-in-progress - I'll post updates from time to time
Also note: this wasn't written in the order it appears here, so if you sense there's a loose end dangling somewhere, you're probably right

e-mail me if you have a question, shawn.charland@skyindustries.com

Background:

If you're like me, you can read the same thing 17 times in 17 different URLs and it doesn't get into your head, but on the 18th read… something happens, and suddenly it makes sense. Hopefully this post will help people who, like me, have a primitive understanding of programming, but still need to do a translation of legacy Delphi code to prism.

Today, I need to convert ~130,000 lines of Delphi code to PRISM, so it will run as an app on a cell phone, or on test equipment that runs Windows CE. At the same time, I need to maintain the Delphi version, for development and other reasons. At the end of the day, I need to living versions of the software: one that runs Delphi, and one that runs in PRISM. The development will continue to be done in Delphi, but I want the latest Delphi version to compile quickly and easily in PRISM.

But there's a problem in my case: back in the day (first half of the 1980s), I was taught "procedural programming", not Object Oriented Programming (OOP). For various reasons, I was able to limp along with procedural programming (with a bit of OOP thrown in), but now it's time for a big change.

A Couple of Good Reference Sites:

I like Prestwood Boards website, loads of good info here: http://www.prestwoodboards.com/ASPSuite/KB/CrossRef.asp?LangID=13&GroupID=

Also there are loads of examples here showing how to use .NET methods to do a selection of common operations:
http://msdn.microsoft.com/en-us/library/ms404278.aspx

Of course, typing your issue into Google will turn up tons of stuff, a sword that cuts both ways.

My Take on the Basic Delphi-PRISM Situation:

1) PRISM let's you keep a lot of your Delphi code, but some Delphi things just don't exist in PRISM, and 2) PRISM lets you an "all access" pass (my term) to the Minotaur's maze (also my term) of .NET methods and properties. .NET is a grab bag full of methods and properties that manipulate files.

If you write code that uses calls to .NET methods and try to compile it in Delphi, you get a truck load of error messages. Compile the same code in PRISM and hey presto, the .NET methods are recognized, and they work as advertised. I love writing in Delphi because (a) Embarcadero's IDE is a dream to use, and (b) I find the language very flexible and intuitive, and natural. Moving to PRISM and relying on .NET methods will be a big adjustment for Delphi programmers I think. To me it's like the difference between building a house using a bunch of really helpful high-level house-building-oriented tools like a bandsaw, a staple gun, radial arm saw, forklift, etc. (that's Delphi), and building a house with a huge selection of low-level tools like screwdrivers, a hammers, crescent wrenches, chisels, files, etc. (that's .NET). In fact, the .NET tool set is so basic it makes sense to use them to build some more complex tools before getting down to business - like using your crescent wrenches and hacksaw and screwdrivers and whatnot to build an electric saw first.

The Rosetta stone for finding and using .NET 4.0 methods and properties is here: http://msdn.microsoft.com/en-us/library/gg145045.aspx
Peruse SYSTEM namespace to get started, you'll get the idea, lots of useful methods in there. BTW in PRISM, the reserved word "unit"" is replaced with "namespace". No idea why. Probably a ploy by MS to sidestep someone's patent, or just generally make things harder. Trust me, you get used to it.

My experience in moving code from Delphi to PRISM is that there may not be a 1:1 mapping between Delphi syntax and PRISM syntax; sometimes there is, and sometimes there isn't. Put another way: there may be several different ways to do "Delphi things" in PRISM, depending on how you decide to use the .NET methods as building blocks.

Procedural Programming vs. OOP:

I come from an ancient "procedural programming" background. PRISM is categorically Object Oriented, full stop. You will have to convert your procedural code to use Classes instead of global variables before proceeding with the conversion. This. Is. Key. You can't fake your way around it, this is 2012.

Global Variables:

Not allowed! But there is a workaround (last paragraph below).

How come "not allowed": as I understand it, PRISM is built for multi-threading so controlling access to global variables when there are a bunch of different threads grabbing for them is just not going to happen. One possible solution for converting legacy Delphi code that uses global

variables: declare a static class that has the (formerly) global variables in it, then instantiate the class when the program starts. That's what I did. If you think about it, it's not even cheating.

Workaround: there is another option, I'm not sure what it does or how well it works, but... in MS Visual Studio, from the main menu choose "Project", then "Project Properties..."; a new tab appears. From the tabs on the right hand side, click on "Compatibility", and check checkbox "Allow globals".

PRISM equivalent of Delphi ASSIGNFILE ( ):

There isn't one exactly; the "equivalent" depends on what you're going to do with the file.

If you're going to read from the file, then here's the syntax in PRISM (without a doubt there are several ways to do this):

```
VAR MyHDFile : StreaWriter;
```
MyHDFile := new StreamReader(<path and file name goes here as a string>);
(SYSTEM.IO.StreamReader is here: http://msdn.microsoft.com/en-us/library/system.io.streamreader.aspx)

This is the equivalent of the following in Delphi"
```
VAR MyHDFile  : TEXT;
ASSIGNFILE(MyHDFile ,<path and file name goes here as a string>);
RESET(MyHDFile);
```

If you're going to write to the file, then here's the syntax in PRISM:

```
VAR MyHDFile : StreamWriter;
MyHDFile := new StreamWriter(<path and file name goes here as a string>);
```
(SYSTEM.IO.StreamWriter is here: http://msdn.microsoft.com/en-us/library/system.io.streamwriter.aspx)

This is the equivalent of the following in Delphi"
```
VAR MyHDFile  : TEXT;
ASSIGNFILE(MyHDFile ,<path and file name goes here as a string>);
REWRITE(MyHDFile);
```

PRISM equivalent of Delphi RESET():

- amazingly, it takes three lines of code to reset the position in a file you're reading from (http://occasional-solipsism.blogspot.com/2006/06/resetting-streamreader.html):

```
MyHDFile.DiscardBufferedData();
MyHDFile.BaseStream.Seek(0, SeekOrigin.TBegin);
MyHDFile.BaseStream.Position := 0;
```

PRISM equivalent of Delphi REWRITE():

- no equivalent; "REWRITE()" effectively happens automatically when you create a new instance of StreamWriter (see equivalent of ASSIGNFILE() above). If there's a way to reposition the pointer in a file you're writing to without closing the file first, I haven't found it.

PRISM equivalent of Delphi READLN():

VAR StringVariable : STRING;
StreamReader.READLINE(StringVariable) reads one line of text into StringVariable. Key point: it reads the ENTIRE LINE. If you want to extract a DOUBLE or INTEGER or whatever from somewhere in that line, as far as I know you have to handle it one character at a time.

PRISM equivalent of "+" String Operator:

You can't use this in PRISM (or .NET): no catenating strings with a "+" sign, unfortunately. Here's a reference: http://wiki.oxygenelanguage.com/en/Delphi_vs._Oxygene, scroll down to "Strings".

Instead, use the SYSTEM.STRING.Join() method (http://msdn.microsoft.com/en-us/library/dd783876.aspx, there are bunch of "Join" methods defined here: http://msdn.microsoft.com/en-us/library/system.string.aspx; each one is called ".Join()", but the interface parameters are different, depending on what you want join). It works like this:
MyCatenatedString := STRING.Join(<separator>,<string01>,<string02>,<string03>, etc.);

THIS IS IMPORTANT! because in PRISM you can't do the Delphi-thing of writing Doubles and Integers and text into a file in a single call to Delphi's WRITELN(); instead, you have to create the entire line as a single string, then write that to the file. SO: convert each number (or whatever) you want written on a single line (see section below re. how to convert stuff, Other Know-How Stuff, "Some Type Conversions"), then catenate them all together using .NET's .Join() method. The StreamWritter method WRITELINE() (see section below, "PRISM Equivalent of Delphi WRITELN") writes the string into a single line, and that's it for that line. As far as I know.

PRISM equivalent of Delphi READ ( ):

- there isn't one exactly; StreamReader.READ() seems to read one character at a time. Here's the MSDN reference: http://msdn.microsoft.com/en-us/library/ath1fht8.aspx

PRISM equivalent of Delphi WRITELN ( ):

Too easy: StreamWriter has a method called .WRITELINE(). Example syntax:

```
VAR MyHDFile      : StreamWriter;
VAR OneLineOfText : STRING;
MyHDFile := new StreamWriter(<path and file name goes here as a string>);
OneLineOfText := 'Here is one line of text'';
MyHDFile.WWRITELINE(OneLineOfText);
```

This writes "Here is one line of text" into one line (the next line) the specified file. If you have half a file built and want to start over, I think you have to close the file (see sections below, FLUSH and CLOSE) and reopen it; closing and reopening definitely works, but there might be a more elegant way.

PRISM equivalent fo Delphi WRITE ( ):

PRISM equivalent fo Delphi FLUSH ( ):

Delphi `FLUSH(MyHDFile) -> MyHDFile.Flush;`

- watch out: use this for StreamWriter, but NOT for StreamReader

PRISM equivalent fo Delphi CLOSEFILE ( ):

Delphi `CLOSEFILE(MyHDFile) ->` PRISM `MyHDFile.Close;`

PRISM equivalent of Delphi EOF ( ):

[Notebook 07 p. 115]

- doesn't exactly exist, but this will work (http://www.codeguru.com/forum/showthread.php?t=432611):

```
WHILE ExternalFile_SpectralRecording_READ.Peek() > 0 DO
```

- "Peek()" looks at next character without "consuming" it; I think of StreamReader and StreamWriter as working like a read/write head that moves stepwise along a tape.

PRISM equivalent of Delphi EOLN ( ):

- don't know, probably have to use combination of StreamReader.READ() and StreamReader.Peek()

PRISM equivalent of Delphi SYNCHRONIZE() method (also how to get a PRISM worker thread to update UI components on the fly):

I can't imagine a more basic function in PRISM (or any other language): the user needs to see what the program is doing on a continuous basis, and forms are used to show this either as changing progress bars, or changing colours on a panel, or changing label captions, or whatever. The

good news: this one is easy (although you do need to be careful of one thing, mentioned at the end). Use Application.DoEvents(), here's how:

Suppose you have a worker thread in Delphi, and a procedure in the worker thread that reads from or writes to properties of a visual component on a form. In order to access or change the properties, you have to use the SYNCHRONIZE method to call the procedure in the worker thread, called like in the worker thread:

```
SYNCHRONIZE(MyThreadProcedure_or_Method);
```

Now suppose in PRISM you have the same situation: a worker thread that reads from or writes to user interface (UI) properties (or any other variable used by the main thread, i.e. the thread that looks after the forms and visual components). In the worker thread, write the code however you want, and after that code insert the line " `Application.DoEvents();` ", just like this. This line works like SYNCHRONIZE in Delphi; it causes the UI thread to update the visual components. If your worker thread has a loop in which UI components are updated in each loop pass, stick `Application.DoEvents()` at the bottom of the loop, and the forms will be updated each loop pass.

Here's an example of a progress bar and editbox that are updated by a worker thread, leaving nothing to the imagination so pardon the simplicity of the example (also I mark my code with lines of stars `******` at the start of procedures/functions to improve readability):

```
{
*************************
*************************
*******  ON-CLICK  *******
*************************
*************************}
method MainForm.button_Start_Worker_Thread_Click(sender: System.Object; e: System.EventArgs);
VAR I : INTEGER;
begin
   {there's a button on the main form that starts the worker thread running}
   INVOKE(@Worker_Thread);{start worker thread}
end;
{
*********************************************************************
*********************************************************************
*********************************************************************
*********************************************************************}
PROCEDURE MainForm.Worker_Thread;
VAR Counter_LOCAL : INTEGER;
```

```
{
*****************************
*****************************
*****************************
*****************************}
PROCEDURE UpdateMainForm;
BEGIN
    {here's the magic call that changes all the UI components on
    the main form, as soon as it's called}
    System.Windows.Forms.Application.DoEvents();
end;
{* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                MAIN BODY OF PROCEDURE Worker_Thread
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *}
BEGIN

   textBox_TestThreadToControlColour.Text := 'THREAD IS RUNNING';
   UpdateMainForm;{show new text in main form editbox}
   Counter_LOCAL := 0;
   WHILE Counter_LOCAL < 100 DO
   BEGIN
      Counter_LOCAL := Counter_LOCAL+1;
      THREAD.Sleep(50);{pauuse for 50 ms else thread runs too fast}
      {increment progress bar by one step}
      progressBar_MainSolutionLoop.Value := Counter_LOCAL;
      {change text in textbox; just show index number (FYI every variable
      in PRISM has a ".ToString" method automatically attached to it)}
      textBox_TestThreadToControlColour.Text := Counter_LOCAL.ToString;
      UpdateMainForm;
   END;{of loop to increment counter}
   {change text in editbox when thread finishes}
   textBox_TestThreadToControlColour.Text := 'THREAD IS DONE';
   UpdateMainForm;
   THREAD.Sleep(1000);{wait 1 second}
   {reset progress bar position to zero}
   progressBar_MainSolutionLoop.Value := 0;
   UpdateMainForm;
```

```
END;
```

The thing you need to be careful of: not sure how to put it into words, but in the example above, if you click the main form button a whole bunch of times (e.g. before the worker thread's main loop has finished executing even one time), the program will restart the worker thread for every button click, sometimes even before the main thread has finished (or more probably each click starts a new instance of the worker thread running, and they're all grabbing for the UI whenever they want, which leads to:

Have a look at the .Lock and .Unlock methods on controls that are manipulated by a thread. I think these are used to prevent collisions when two threads might try to access the same visual component at once. Probably you have to lock all controls that the thread uses when it wants to access them, then unlock them when the thread is done with them.

Let's say you have a thread that's manipulating (or might manipulate) a graphical component like a progress bar; before changing one of its properties, I think the thread needs to call the .Lock method for the component, do its thing, then call .Unlock, like this in PRISM:

```
MyProgressBar.Lock;
MyThreadProcedure_or_Method;
MyProgressBar.Unlock;
```

I already do a bit of the Lock/Unlock thing in Delphi, whenever I'm drawing on a canvas for example. I'm out of my depth writing about this, just want to share some thoughts.

Other Know-How Stuff:

Local Variables:

- watch out for duplicate-name local variables inside methods inside another method; not allowed (http://wiki.oxygenelanguage.com/en/Delphi_vs._Oxygene, scroll down to "Local Variable")

The VAR Issue:

- VAR caused me problems - it works in the usual Delphi way in declarations, but if you call a method that has a VAR variable at its interface, you need to put VAR *in the call*. Here's what the problem looks like:

> This procedure definition compiles fine in both Delphi and PRISM:
>
> ```
> PROCEDURE DoSomething(InputVariable 01_LOCAL, IputVariable02_LOCAL : INTEGER; VAR
> SecondVariable_LOCAL : DOUBLE);
> ```

The Delphi call looks like this:

```
DoSomething(InputVariable 01, IputVariable02,SecondVariable);
```

This line will generate an error in PRISM.

The PRISM call has to look like this:

```
DoSomething(InputVariable 01, IputVariable02,VAR SecondVariable);
```

Notice the VAR in the call syntax. This will compile and run fine in PRISM.

To solve this: in MS Visual Studio, from the main menu choose "Project", then "Project Properties..."; a new tab appears. From the tabs on the right hand side, click on "Compatibility", and check checkbox "allow implicit var/out in method calls".

I also check "Allow Delphi compatibility syntax" and "Use Delphi compatible division operators".

String Indices:

- first index in Delphi is [1]; first index in PRISM is [0], like this:

```
MyString : STRING;
MyString := 'Ding!";
```

–> in Delphi, MyString[1] = D
–> in PRISM, MyString[1] = i, MyString[0] = D
–> in Delphi, MyString[0] = compiler error

Some Type Conversions:

To convert from String to DOUBLE: Variable_DOUBLE := `CONVDERT.TODOUBLE(MyString);`

How to Start a Thread:

I think there is more than one way to do this, but this seems to work (see example in section above about SYNCHRONIZE()):

```
{*********************************}
method MainForm.button_Start_Worker_Thread_Click(sender: System.Object; e: System.EventArgs);
VAR I : INTEGER;
begin
   {there's a button on the main form that starts the worker thread running}
   INVOKE(@Worker_Thread);{start worker thread}
```

```
end;
{*******************************}
PROCEDURE MainForm.Worker_Thread;
BEGIN
    {do stuff here}
END;
```

CF Application Development in PRISM:

This is what I'm interested in right now. Here's the story, from the Oxygene Wiki: http://wiki.oxygenelanguage.com/en/Compact_Framework_support . (version I see when writing this: last modified 19 Oct 2008).

Turns out the stripped-down version of Visual Studio 2010 (Visual Studio Express) shipped with PRISM/RAD Studio won't do it, you need the full version (as in, $$$) of MS Visual Studio to do this (90 day trial versions are here: http://www.microsoft.com/visualstudio/en-us/try). I installed the trial version of MS Visual Studio 2010 Pro, which comes with Windows SDK for .NET 3.5 (I think it's 3.5). Surprising discovery way downstream: apparently you can't do Windows CE development with Visual Studio 2010; the latest version of Visual Studio that supports this is Visual Studio 2008. I'll leave the next part (below) in this text since there might be some readers interested in the Visual Studio 2010 Pro route for non-CF development.

- -

This next part is a monologue about what I did installation-wise, starting with recommendations from Embarcadero Support, to illustrate the dead ends I looked at and why and how. I ended up doing what the paragraph above says, so if you don't care how I got there, you can skip this next part, down to where the dashed marker line is.

Embarcadero Support says that as well as the full version of (at least) the Standard version of Visual Studio, you also need the Compact Framework Software Development Kit (SDK) for .NET 2.0, and they also recommend getting Power Toys for CF 3.5, both are found here: http://msdn.microsoft.com/en-us/netframework/aa497280.aspx . I notice there's a numbering disconnect between Power Toys and SDK for .NET 2.0, does it matter? Don't know yet, but I did notice that on this page, right below "Overview", there's this text:

> *NOTE: This version of the .NET Framework SDK SDK has been superceded by the Windows SDK for Windows Server 2008 and .NET Framework 3.5.*

I asked Embarcadero Support about this; here's their reply, including bit of bad news about graphical form design in PRISM:

> Delphi Prism supports .NET Compact Framework 1.0 and 2. This is why I referred you to the .NET SDK 2.0. The .NET 3.5 SDK is supposed to include all of the things in the 1.0 and 2.0 versions so it might work for you as well. To answer your previous email question, there is no Compact Framework designer support in Prism. Microsoft has tied the Compact Framework designer support closely with C# and VB and the hooks in Visual Studio are not available to us to hook in Prism. You can still create applications but you have to create your controls dynamically via code.

Following the SDK story a bit farther, for anyone who's interested the download site for "Windows SDK for Windows Server 2008 and .NET Framework 3.5" is here: http://www.microsoft.com/download/en/details.aspx?id=11310
Naturally, there are a bunch of warnings not to download this unless you really really want to; here's what it says at the top of this page (publishing date looks like 2/5/2008):

> *Installing the newly released Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 SP1 instead of this release is recommended. If you do go ahead and install this SDK after VS2008 SP1, please ensure the patch described in Knowledge Base 974479 is applied. See Overview section for more information.*

Grrrr.

So here's the download site for Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 SP1:
http://www.microsoft.com/download/en/details.aspx?id=18950

Here's what it says on that page, under Overview:

> ***Overview***
>
> *The Windows SDK for Windows 7 and .NET Framework 3.5 SP1 provides the documentation, samples, header files, libraries, and tools (including C++ compilers) that you need to develop applications to run on Windows 7 and the .NET Framework 3.5 SP1. To build and run .NET Framework applications, you must have the corresponding version of the .NET Framework installed. This SDK is compatible with Visual Studio® 2008, including Visual Studio Express Editions, which are available free of charge.*
>
> *Please see the Release Notes for the full list of supported platforms, compilers, and Visual Studio versions and any late breaking issues. For detailed information about the content in this SDK, including a description of new content, please see the Getting Started section in the documentation.*

System requirements

Supported operating systems: Windows 7

Windows 7; Windows Server 2003; Windows Server 2008; Windows Vista; Windows XP Hard disk space - choose a disk drive with a minimum of 2.5 GB of free space. We highly recommend that you have your machine fully patched through Microsoft Update before beginning setup. The .NET Framework 3.5 SP1 tools and samples contained in this SDK require installation of the .NET Framework 3.5 SP1 which is available with Visual Studio 2008 or a standalone install of the .NET Framework 3.5 SP1.

At least there's no warning this time. For some reason, these things are never straightforward. Anyway Embarcadero says to use .NET SDK 2.0... although the others might work.

I started by installing the 90 day trial version of MS Visual Studio 2010 Pro: 1) extract .rar files, 2) burn ISO onto DVD, then 3) running setup.exe from disk, took about 40 min on my Vaio laptop (I have a high speed Internet connection). When I tried to run MS Visual Studio 2010 Pro (MS VS Shell from PRISM installation is still installed on my PC), I got this message:

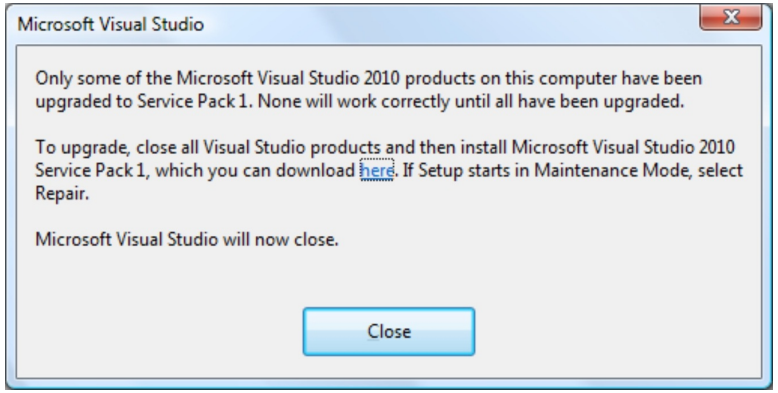


Figure 1: Yet another error message

Here's where Service Pack 1 is: http://www.microsoft.com/download/en/confirmation.aspx?id=23691 . I downloaded the file (VS10sp1-KB983509.exe) and ran it. This starts a 365 MB download, took 1.5 hrs to finish download and installation. As suggested in the "finish" dialog box, I checked for Windows Updates; there were 8 (294.7 MB), which I think are attributable to the newly-installed SP1, since

there were no updates immediately before installing SP1.

BTW the installation dialogue box says “Windows SDK” is installed along with all the other applications; no mention of the .NET version but I think it’s 3.5. I’m didn’t install .NET SDK 2.0 over (suspected) 3.5 since installing v. 2.0 of anything over v. 3.5 seems like an open invitation to the time vampires. And they always accept invitations.

After noticing that the SDK folder mentioned in the Oxygene CF development page (http://wiki.oxygenelanguage.com/en/Compact_Framework_support , from above) doesn’t exist on my PC after my installation of Visual Studio 2010 Pro, I looked at Wikipedia re. MS Windows SDKs: http://en.wikipedia.org/wiki/Microsoft_Windows_SDK ; right at the top it says this (which I colour coded for clarity):

> *Microsoft Windows SDK, Platform SDK, and .NET Framework SDK are software development kits from Microsoft that contain header files, libraries, samples, documentation and tools required to develop applications for Microsoft Windows and .NET Framework.[1]*
>
> *The difference between these three SDKs lies in their area of specialization: Platform SDK specializes in developing applications for Windows 2000, XP and Windows Server 2003. .NET Framework SDK is dedicated to developing applications for .NET Framework 1.1 and .NET Framework 2.0. Windows SDK is the successor of the two and supports developing applications for Windows XP, Windows Vista, Windows 7, Windows Server 2008, .NET Framework 3.0, .NET Framework 3.5, and .NET Framework 4.0.[2] It contains extensive documentation and around 800 samples.*

- -

It looks like PRISM isn’t keeping up with SDK releases, since it’s only qualified to work with .NET SDK 2.0. So I next installed that SDK, opening the door to the time vampires. The time vampire portal is here: http://www.microsoft.com/download/en/details.aspx?id=17981 .

After installing .NET SDK 2.0: no vampires yet, and the folder mentioned on the Oxygene page is now present: C:\Program Files (x86)\Microsoft.NET\SDK\CompactFramework\v2.0\WindowsCE (except the Oxygene page specifies ..\v3.5\.. instead of ..\v2.0\.., and Embarcadero only officially endorses PRISM for .NET 2.0 [secretly they say it probably works with .NET 3.5, but presumably testing isn’t finished]).

The instructions on the Oxygene page say to create a Visual Studio project that’s a “Delphi PRISM class library: - I don’t see that option exactly, but there is an option to create a Mono class library:

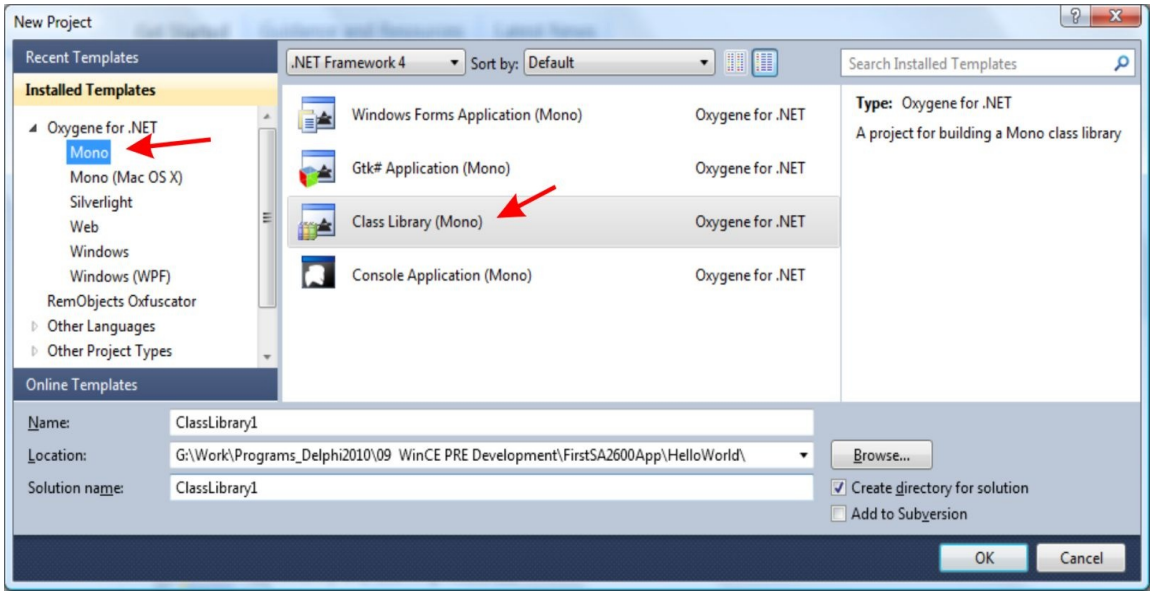
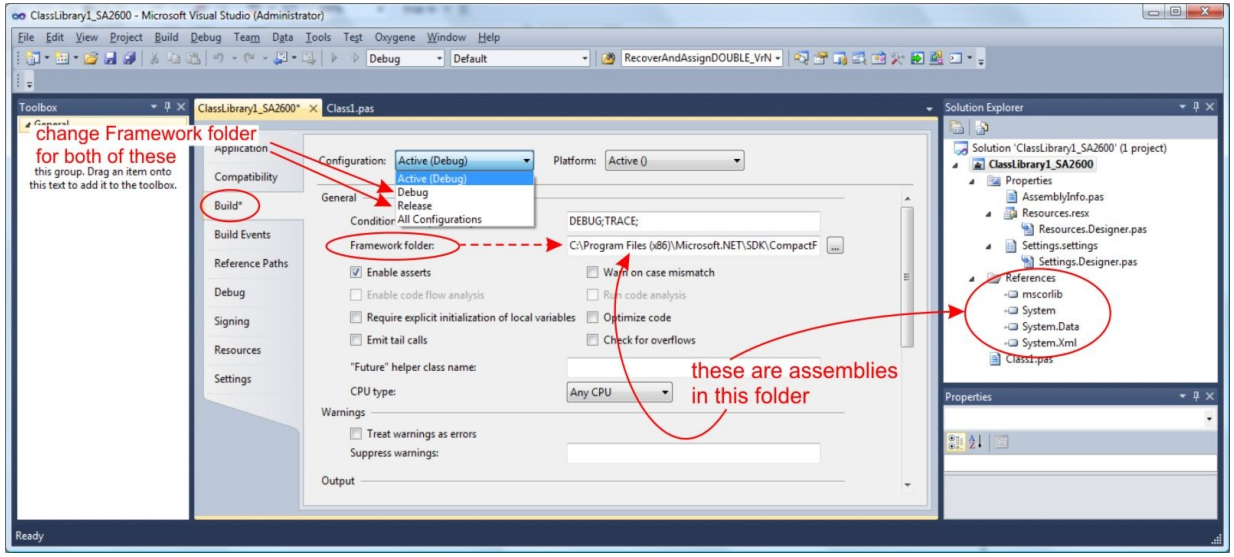


Figure 2: New project: Class library (Mono)

Following the Oxygene page's instructions, it says to go to Project –> Project Properties... and select the "Build" tab (on the left side). Objective: change the Framework folder. But first: Notice the combo box near the top (see image below)? It expands to allow you to select between "debug" and "release" versions. Here's what that means: when you build the solution, the result ends up in a folder called \debug\; in my case, the path to the executable looks like this:

G:\Work\Programs_Delphi2010\09 WinCE PRE Development\FirstSA2600App\HelloWorld\HelloWorld\bin\Debug\HelloWorld.exe

There's another folder called \release\ at the same level as \debug\, so "debug" and "release" versions of your app are directed to different folders. Handy, and it makes sense. BTW in this case ["Class library (Mono)" project] a .dll is produced, along with two other files: .mdb and .pdb. How you direct VS to put the output files in the \release\ I don't know yet, so far I see the .dll etc. files are only in \debug\ folder. The \release\ folder is empty. Anyhow we have to change the Framework folder for BOTH "debug" and "release" - remember we're only making a PRISM .dll here, not a stand-alone app (yet), so change them both.



So: pick the "debug" combo box option, and change the Framework folder, then do the same for the "release" option. BTW: DING! The penny drops and now it's obvious why the folder needs to be changed. The files in

C:\Program Files (x86)\Microsoft.NET\SDK\CompactFramework\v2.0\WindowsCE

also exist - with the same names - in various other Microsoft folders (can't be bothered to find them, have a look yourself). Since the files (assemblies) have the same names, MS has distinguished them by putting them in different folders. Makes sense, handy too; since all the assemblies have the same name, linking to them is a snap... but you have to tell Visual Studio which set you want to use, and that means specifying the assemblies' folder. So that's that. The instructions on the Oxygene page are thin on explanation, but they are starting to make more sense.

There are more project options to set: on the left side of the "Project Properties" form, change from the Build tab to the Compatibility tab. Here's what I've checked off for my project (below). Doesn't mean this is right for your project, it's just what *I've* used.

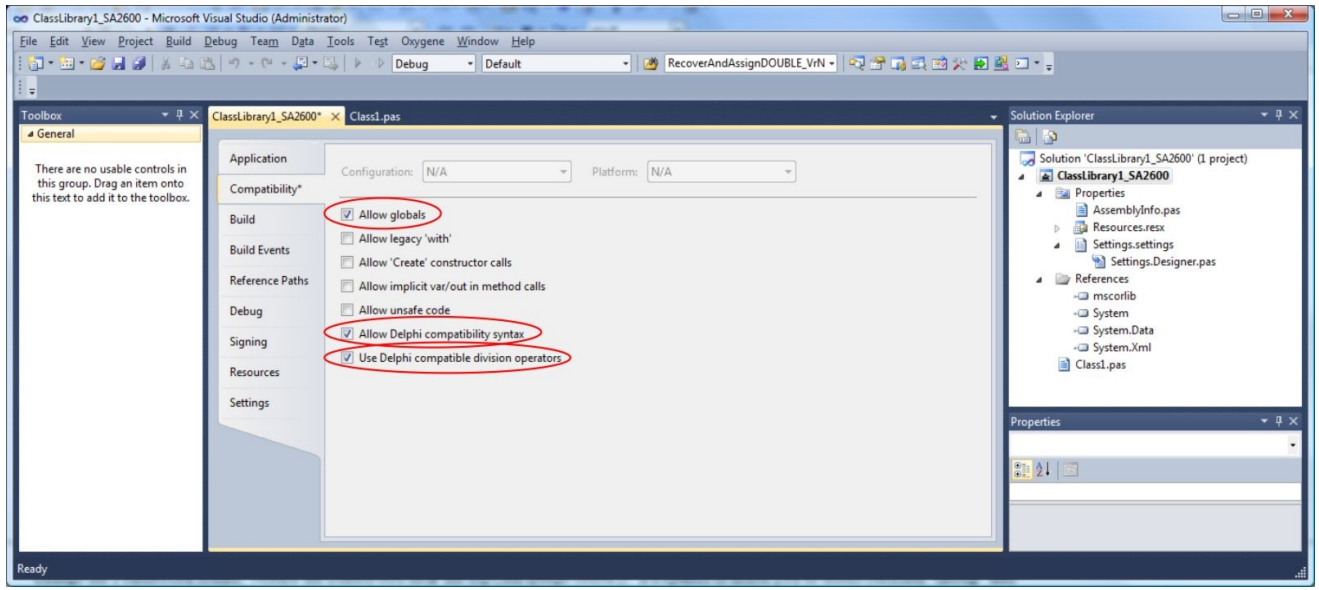


Figure 4: Project compatibility options (for specifically my project)

Now go to the "Application" tab, the top one. *This is not in the Oxygene page instruction set*, but I noticed that the "Target Framework" combo box says, at this point, ".NET 3.5". Well that's got to be wrong; I installed .NET SDK 2.0 per Embarcadero's recommendation. So, I changed it to .NET 2.0 (figure below). Visual Studio will show a warning dialogue box, then close and re-open the solution automatically.

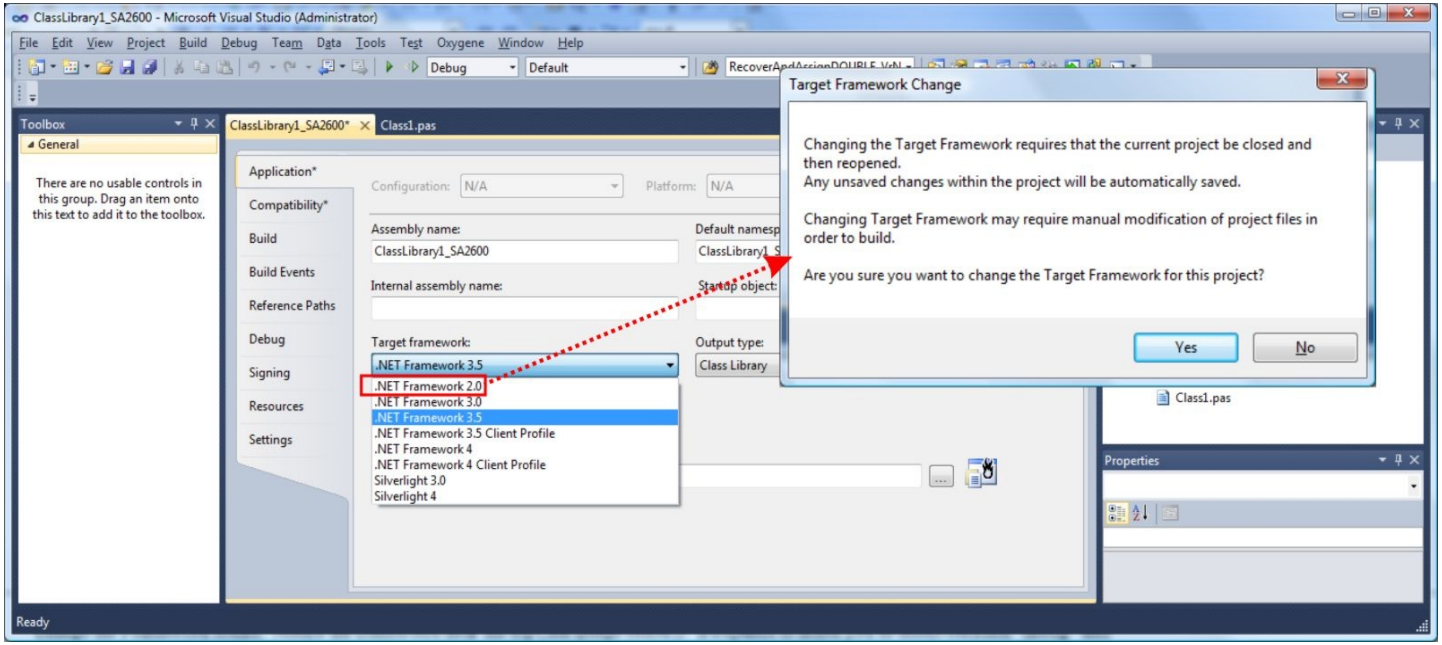


Figure 5: Changing the target framework from .NET 3.5 to .NET 2.0

What IS in the Oxygene page's instruction set is to go to expand the "project references" and replace each of the assemblies marked with a yellow exclamation mark with the same-name assembly in the Compact Framework folder. As they say, do this by right-clicking on "References", choose "add reference...". I don't see any exclamation marks in my references, so maybe changing the Target Framework obviates this step (the Oxygene page I'm working from is dated 19 Oct 2008).

Now on to PRISM code development; since a visual form designer isn't integrated with PRISM (as in, you can't drag-and-drop controls onto a form), you've got two choices:

> Choice 1: Build your PRISM app and add forms and controls manually, and set all the properties of the controls by hand, and test them for appearance and function on the device (or emulator) as you go along. Sounds monumentally painful and slow.
>
> Choice 2: Per the Oxygene page, follow a two-step process (which isn't as bad as it sounds):

Step 1: write your Delphi PRISM code as a class, specifying the methods and properties of the class.

Step 2: create a C# (or VB.NET) app with the visual controls put this-and-such way on a form, go to the Solution Explorer pane in Visual Studio, right click on “References”, choose “add reference...”, and find the PRISM-created .dll (aha, the “library” of classes) and add it (the .dll in the \release\ folder is probably the right one to use). Now you can use the classes you defined in Step 1; instantiate them in the C# (or VB.NET) app, and hey presto, it runs. That’s the theory anyway.

I took Choice 2. I chose a C# Windows Forms Application as a new C# project, then under Project –> Project Properties, selected the “reference paths” tab, and added the path to the .NET 2.0 CF assemblies (C:\Program Files (x86)\Microsoft.NET\SDK\CompactFramework\v2.0 \WindowsCE), as shown below.

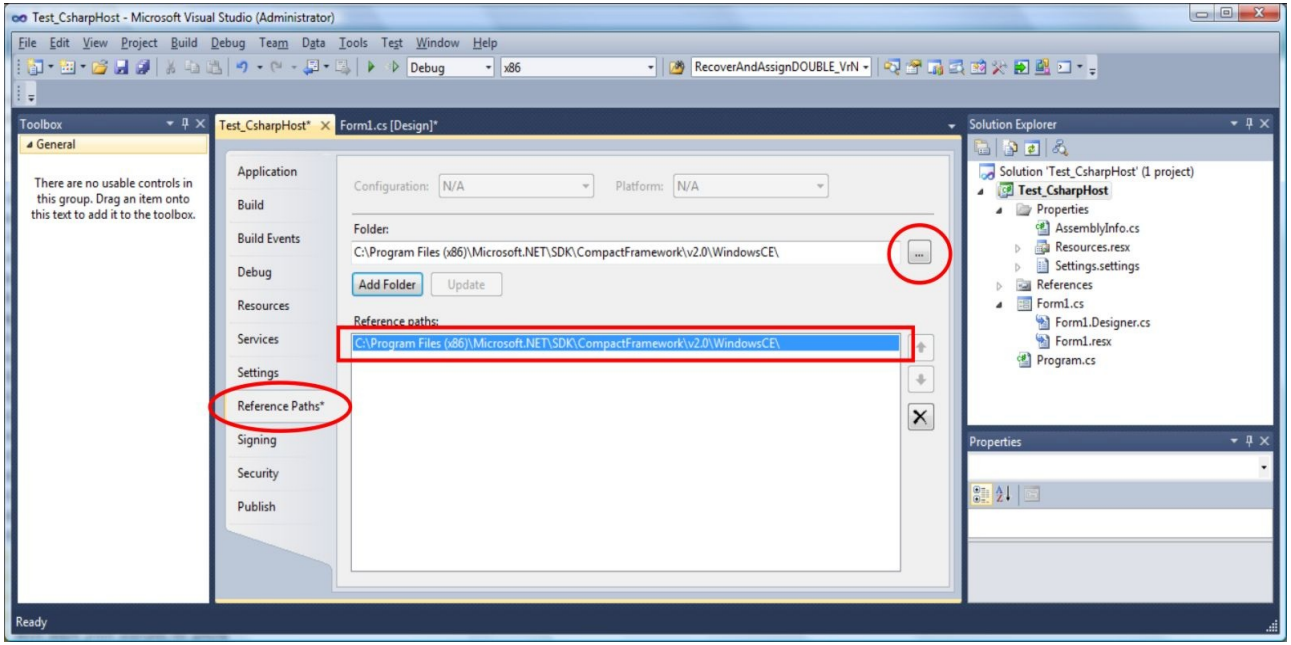


Figure 6: Setting C# reference path to .NET 2.0 windows CE

Not done yet! Go to the application tab, and set the target framework (below).

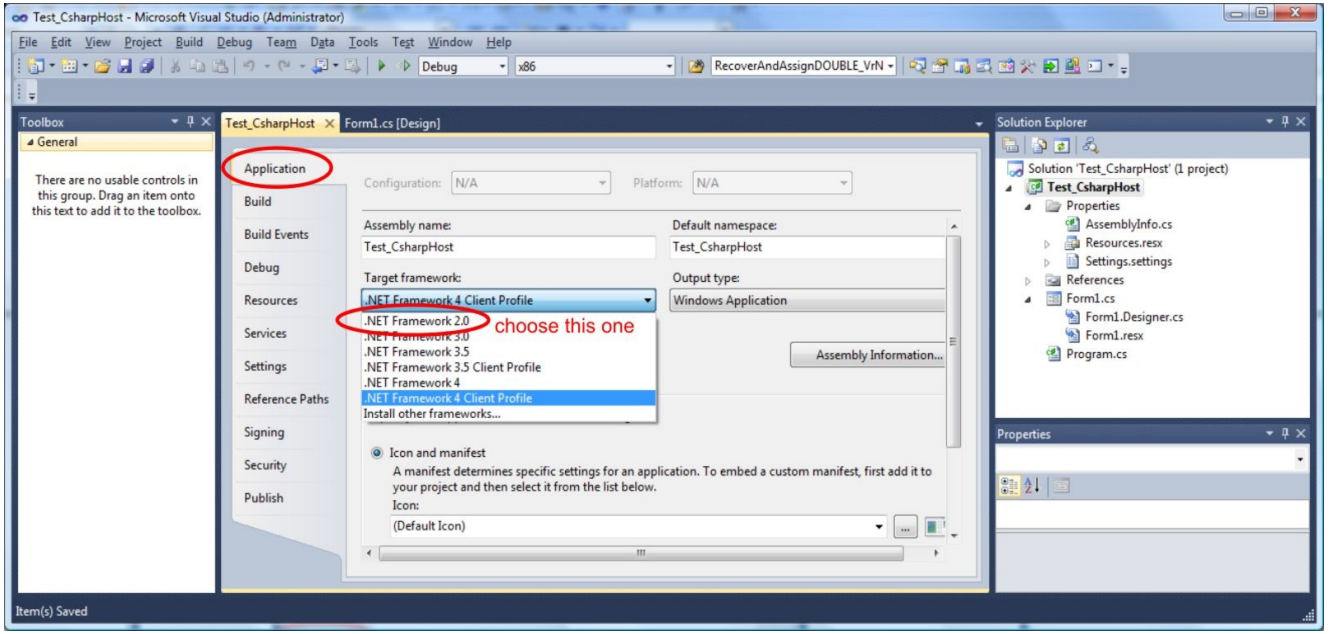


Figure 7: C# host: set the target framework in the application tab

Now there's a form to work with, one that will work in Windows CE. Try putting a button on it to do PRISM stuff.

Ding! an error message appears:

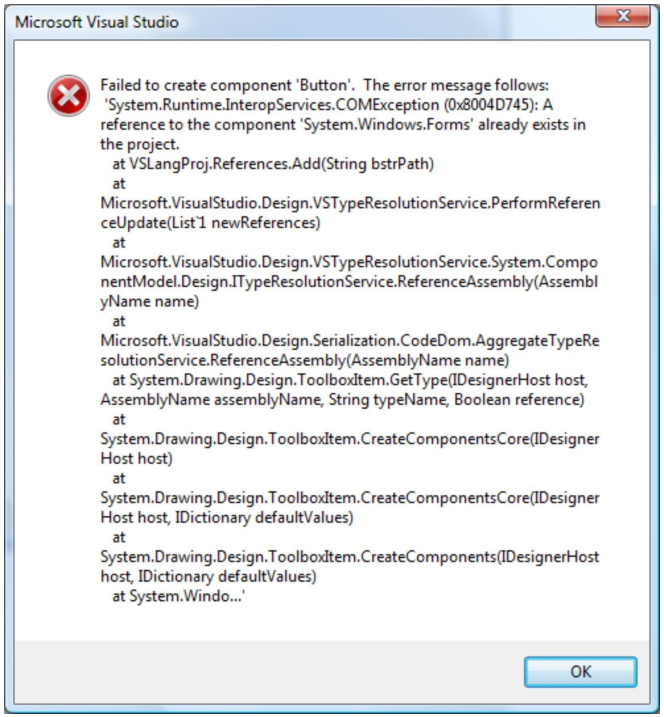


Figure 8: Surprise error message when trying to put button on form

And guess what! After all that, Visual Studio doesn't support Windows CE development! At least according to here: http://stackoverflow.com/questions/2823288/develop-for-wince-with-vs2010 .

Conclusion: I have to downgrade to Visual Studio 2008. Grrrrrrr yet again.

To uninstall the trial version of Visual Studio 2010: http://social.msdn.microsoft.com/Forums/eu/vssetup/thread/e15ba6f9-f23d-4cbb-8032-9610daeb3abd . Warning: the uninstaller whose link is at the top of the URL will also uninstall Visual Studio 2010 Shell that came with PRISM.

Visual Studio 2008 Pro trial version (90 days) is here: http://www.microsoft.com/download/en/confirmation.aspx?id=3713 .

But when I get it working:
Double click on the button to see the onclick event handler. Instantiate one of the classes from the PRISM .dll. Do something with one of the methods in the now-instantiated class. Build the project. Run project on WinCE platform. And of course the C# host needs to read and write data to the methods in the PRISM class, and that will require at least one worker thread so the graphical interface gets updated cyclically, but that's probably pretty straightforward. It'll be in rev-01 of this document.

If It's a Mobile Device Where You Want Your Software to Run:

To deploy the project to a mobile device (that's a *mobile* device, not for example a spectrum analyzer running Windows CE), have to make a .cab file; here's what Microsoft has to say about it: http://msdn.microsoft.com/en-us/library/bb158712.aspx

And here's the best ever reference (by José Gallardo Salazar, http://www.mobilepractices.net/aboutme) on how to do it with Visual Studio 2008: http://www.mobilepractices.com/2008/02/how-to-create-windows-mobile-smart.html

BTW at the step "Now it's time to add the resource file" I discovered there is no .XML file in my project group! So that was a bit of a snag; expert advice (Dave Schmidt, Tektronix) is that this step isn't necessary (advice was not checked since in the end I found I didn't need a .cab file, I'm not targeting a mobile device after all, plus Dave most definitely knows his stuff).

How to embed resources in project (Visual Studio 2008), a necessary step in creating the .cab installation file:
http://support.microsoft.com/kb/319292

Anyway I made a .cab file and deployed it to a spectrum analyzer (i.e. I copied the .cab file onto the spectrum analyzer) and double clicked on it. Zing, worked as advertised. Unfortunately the .exe file wouldn't run. The reason: my project was built to reference .NET 3.5, but *those assemblies are not loaded on the spectrum analyzer, in order to save memory*. Bit of a surprise there, but good to know. So be aware: it's possible for a platform to not have all the assemblies you are expecting IF the manufacturer has a deal with Microsoft to provide them with a custom SDK. This may shoot my project in the head, since I'm planning to use PRISM to create a shared library that uses the .NET Compact Framework, so if the assemblies aren't loaded on the target machine... yeah, not good.

Nevertheless, all this lead to an important piece of information: when building for WinCE, the executable produced by the build will run as-is on the WinCE device, no installer etc. required, just like .exe files run on full-fat Windows. The .exe will also run on the target machine from a shared folder (e.g. Ethernet cable connecting laptop and instrument), no need to copy the executable to the instrument. This may be obvious to everyone else, but in the mouse maze of Visual Studio and .NET, it seemed to me anything is possible... and as we all know, there are one trillion

ways to do something wrong, and usually only one or two ways to do it right.

So back to the story: I still need a UI shell to call the methods in the classes I instantiate from my PRISM-produced .dll, if I ever get that far. In my case the advice was to make a new project with Visual Studio 2008 (fig. 9), and choose Visual C++ | Smart Device | MFC Smart Device Application and click "OK". [BTW you'll see at the top of the form that .NET assembly is referenced, but Super Dave said never mind that, it's not used in MFC (what MFC is: http://www.codeproject.com/Articles/1284/Win32-vs-MFC-Part-I].

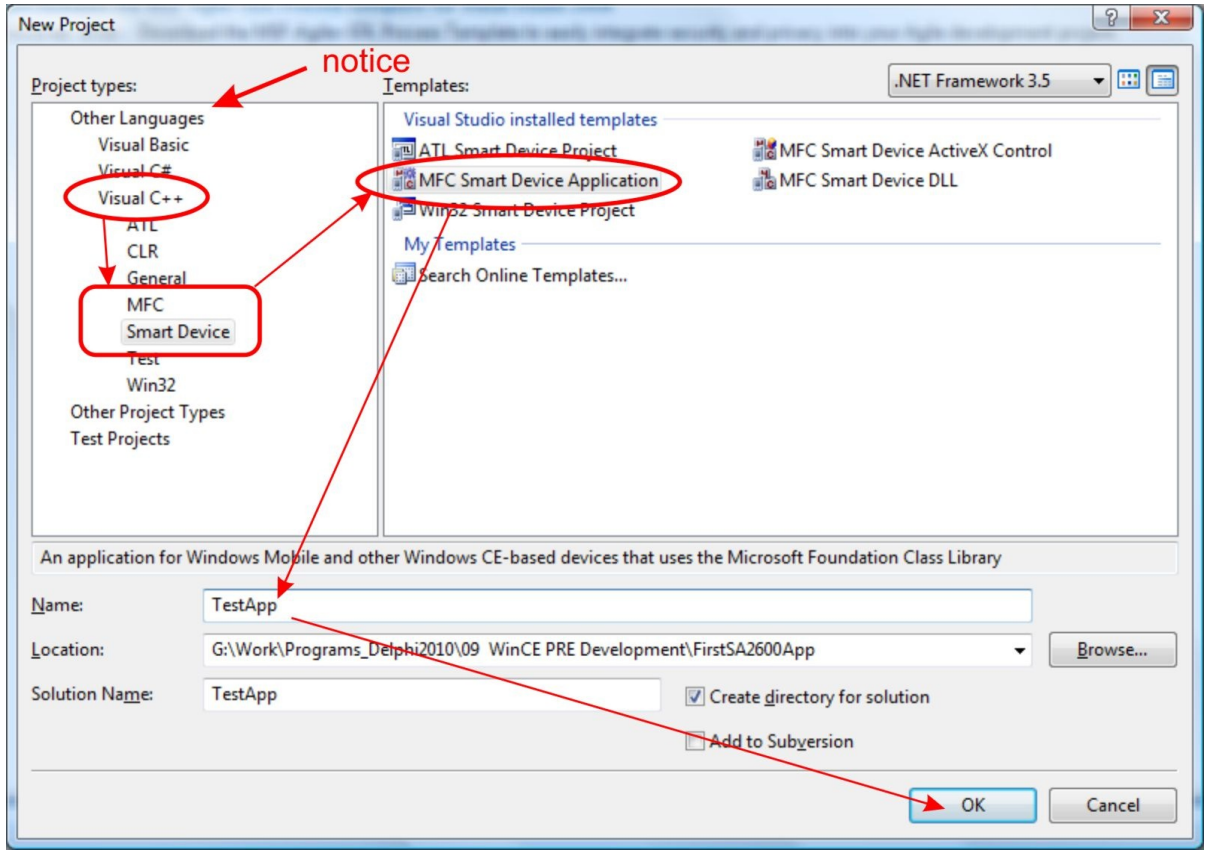


Figure 9: Open a WIn32/MFC project

After you click “OK”, a new dialog box appears, click “next”, and another dialog appears (fig. 10). I’m targeting the standard SDK for WinCE 5.0, so I used the controls between the two panes to move all the SDKs to the left pane, and the only SDK I’m using (STANDARDSDK_500) into the right pane. Click “next”.

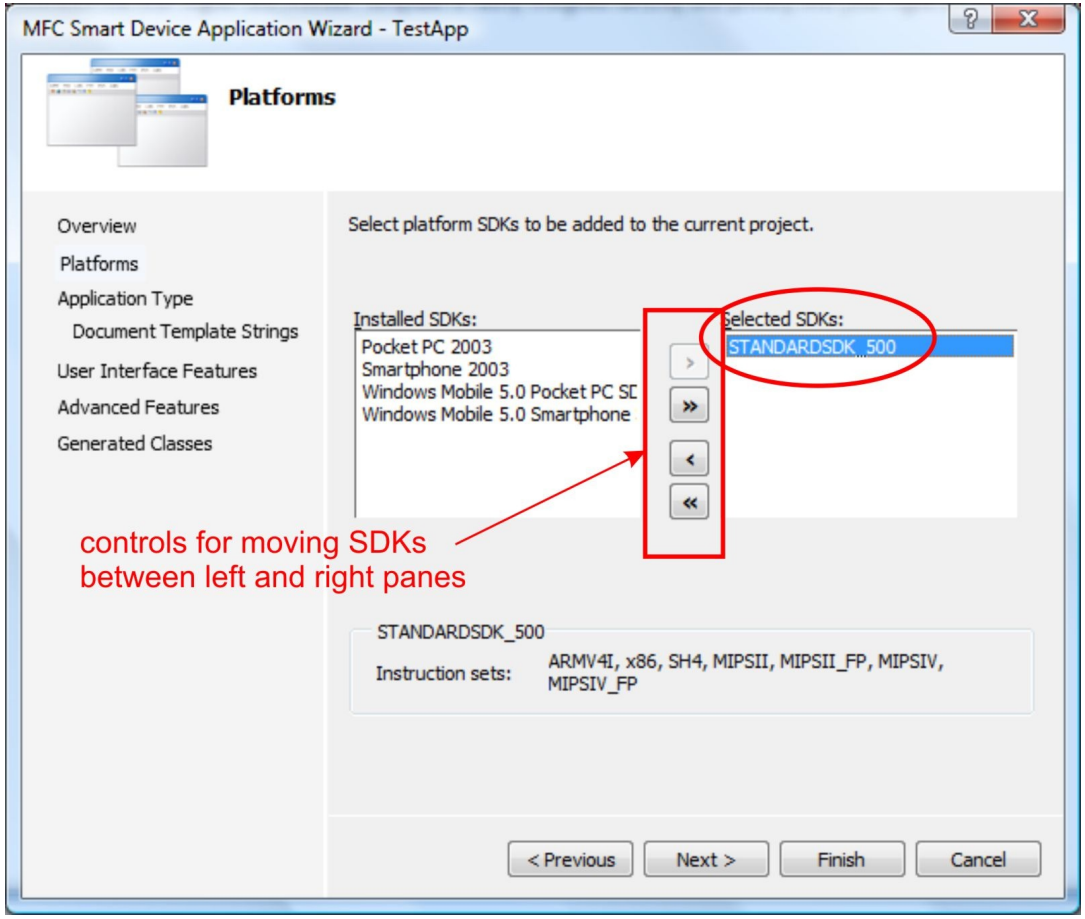


Figure 10: Select the SDKs your app will use

Yet another dialog appears (fig. 11); set Application type to “Dialog based”, set Use of MFC to “Use MFC in static library”. Click “finish”.

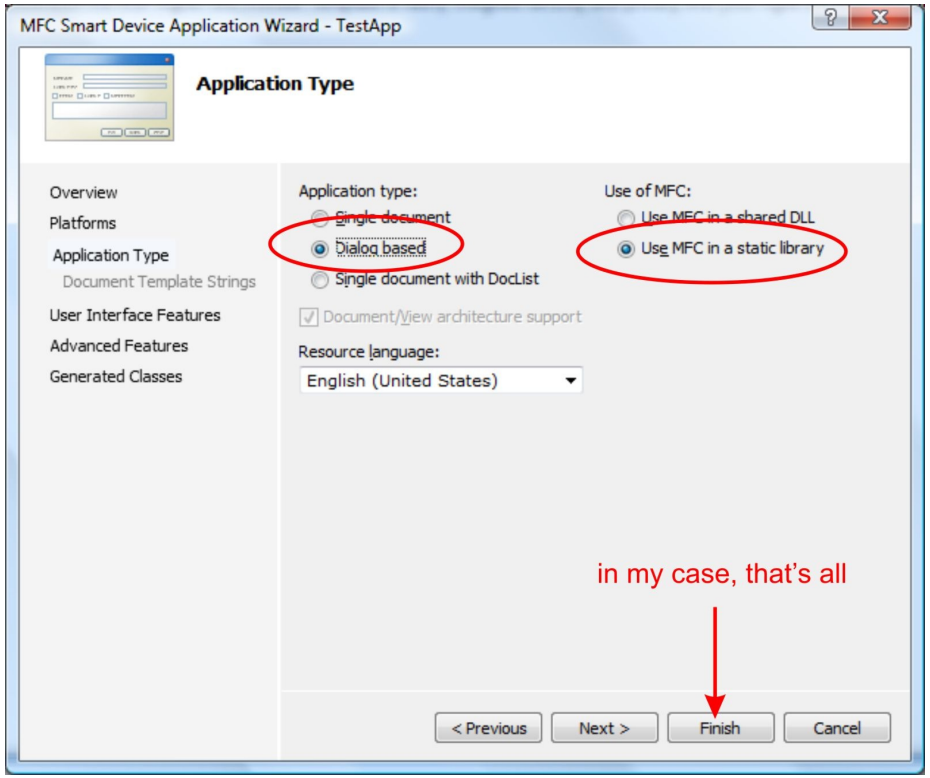


Figure 11: Make MFC app dialog-based

Right away you’ll notice there is no sign of a designer form. Here’s how to make it appear: in the Solution Explorer pane (fig. 12), look in “Resources” and double click on the .rc file.

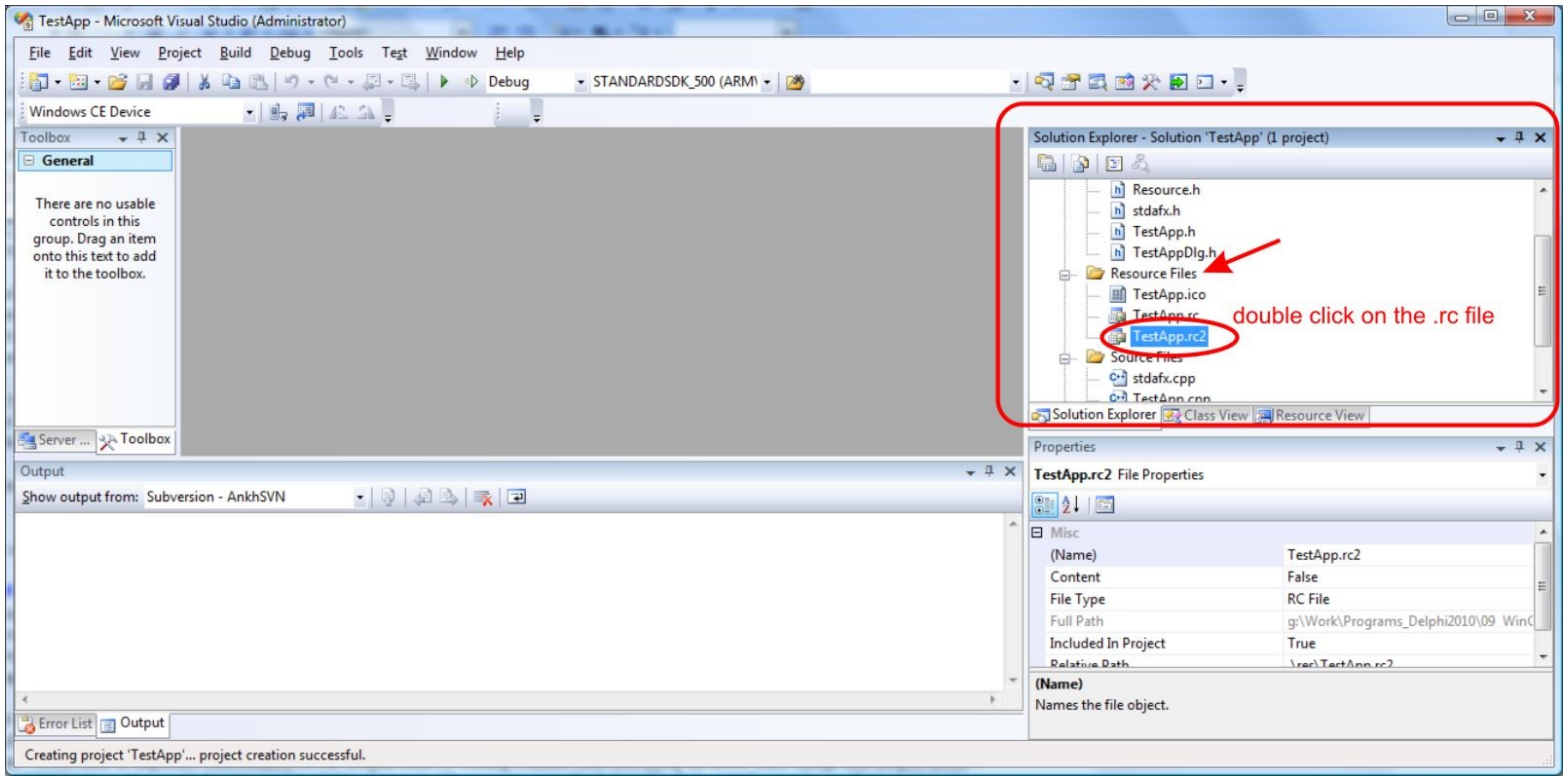


Figure 12: Make the designer form appear - step 1

The view changes (fig. 13); in the Solution Explorer pane you'll now see just the .rc file. Expand the "Dialogs" folder, and double click on the (so far only) file there. Hey presto, the form appears on the left. along with a somewhat restricted collection of visual components and an even more restricted selection of properties for those components (as I found when I started using a few of them).

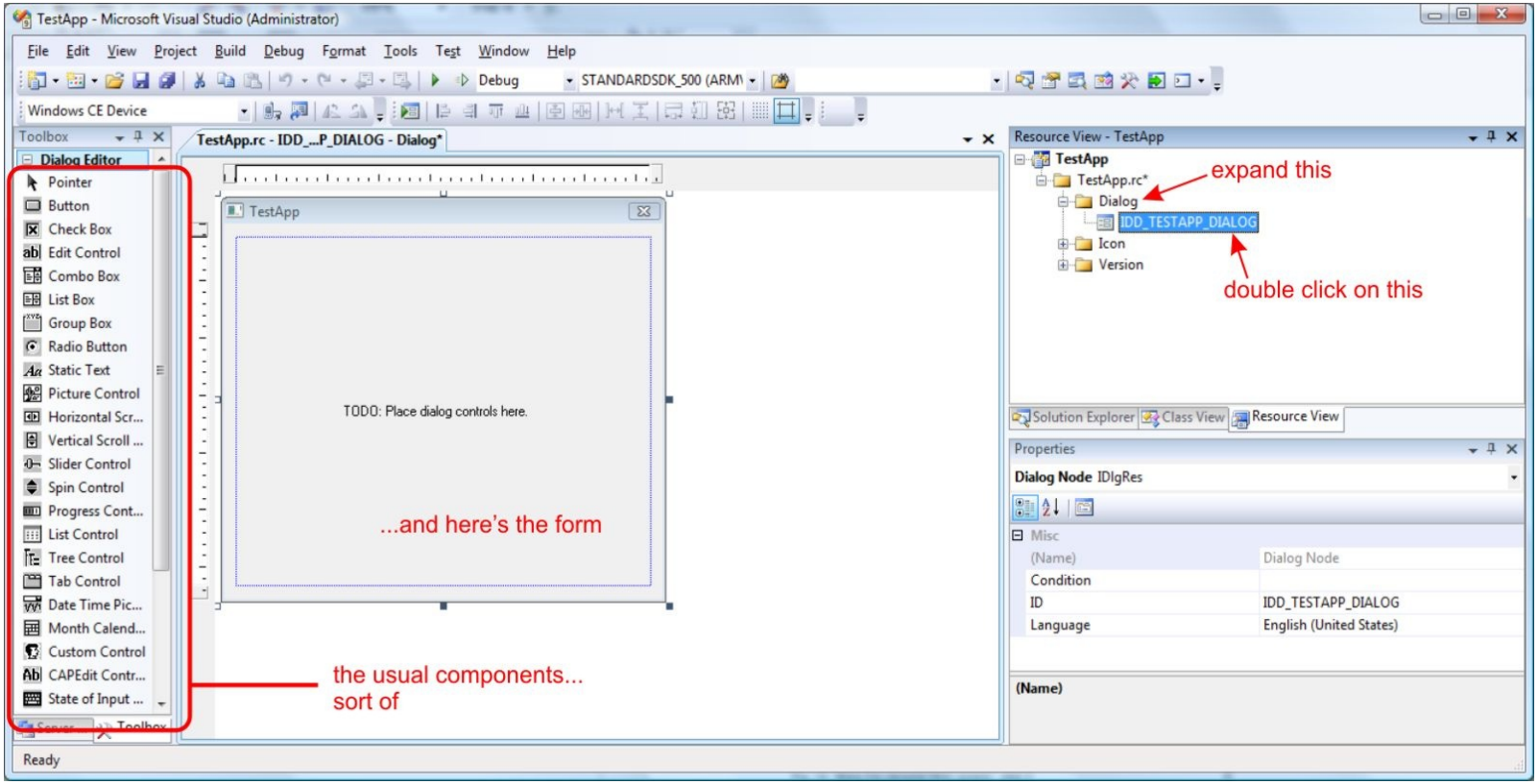


Figure 13: Make the designer form appear - step 2

BTW if you want to add another dialog box, or a bitmap, right click on any of the folders in this pane and from the menu, choose "Add Resource". A new window appears (fig. 14), pick what you want and click the applicable button ("new", "import", "custom").

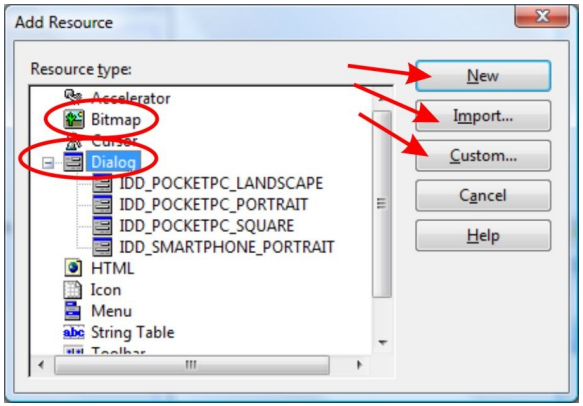


Figure 14: Dialog box to add a resource

If you want to add a bitmap to the form, choose the "picture control" from the tool box. This will make a tiny icon appear on the form which by itself does, in fact, nothing. You have to associate it with a bitmap to get a picture on the form. To do this, you have to add the bitmap (that's .BMP not .JPG or anything else) as a resource as described above. Here's a picture of how to do it, in figs. 14 and 15. It's super-primitive, you have to make the bitmap the right size in pixels, for the form. I use CorelDraw to export graphics as .BMPs, and set the size in pixels in one of the export steps.

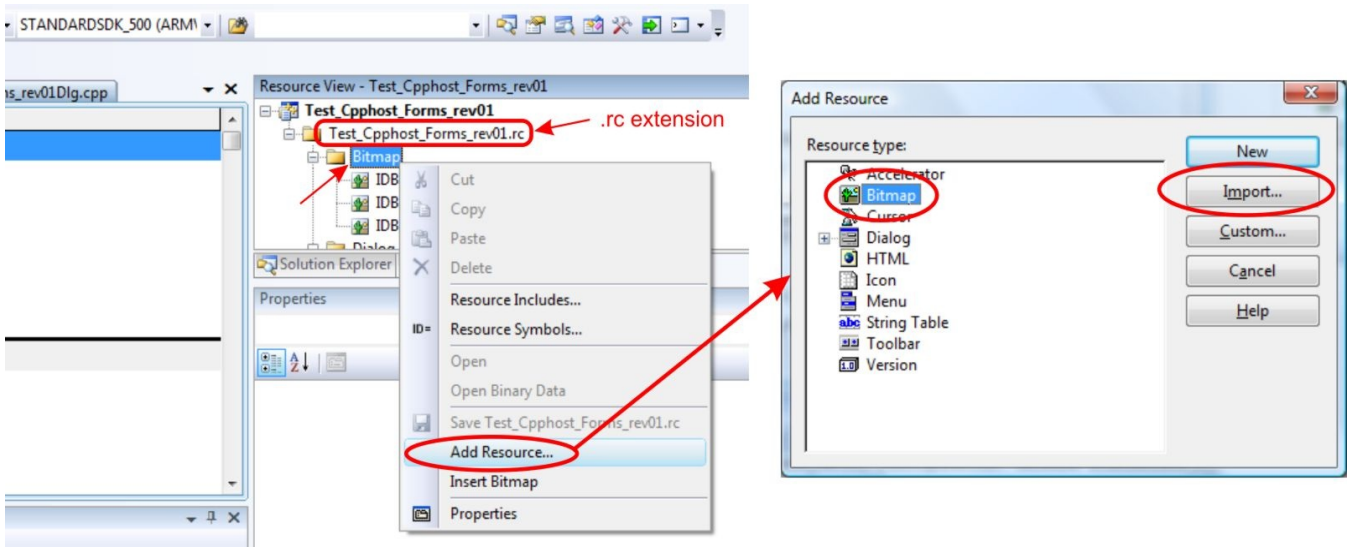


Figure 15: How to add a .BMP file to Visual Studio 2008 Win32/MFC “Picture Control”

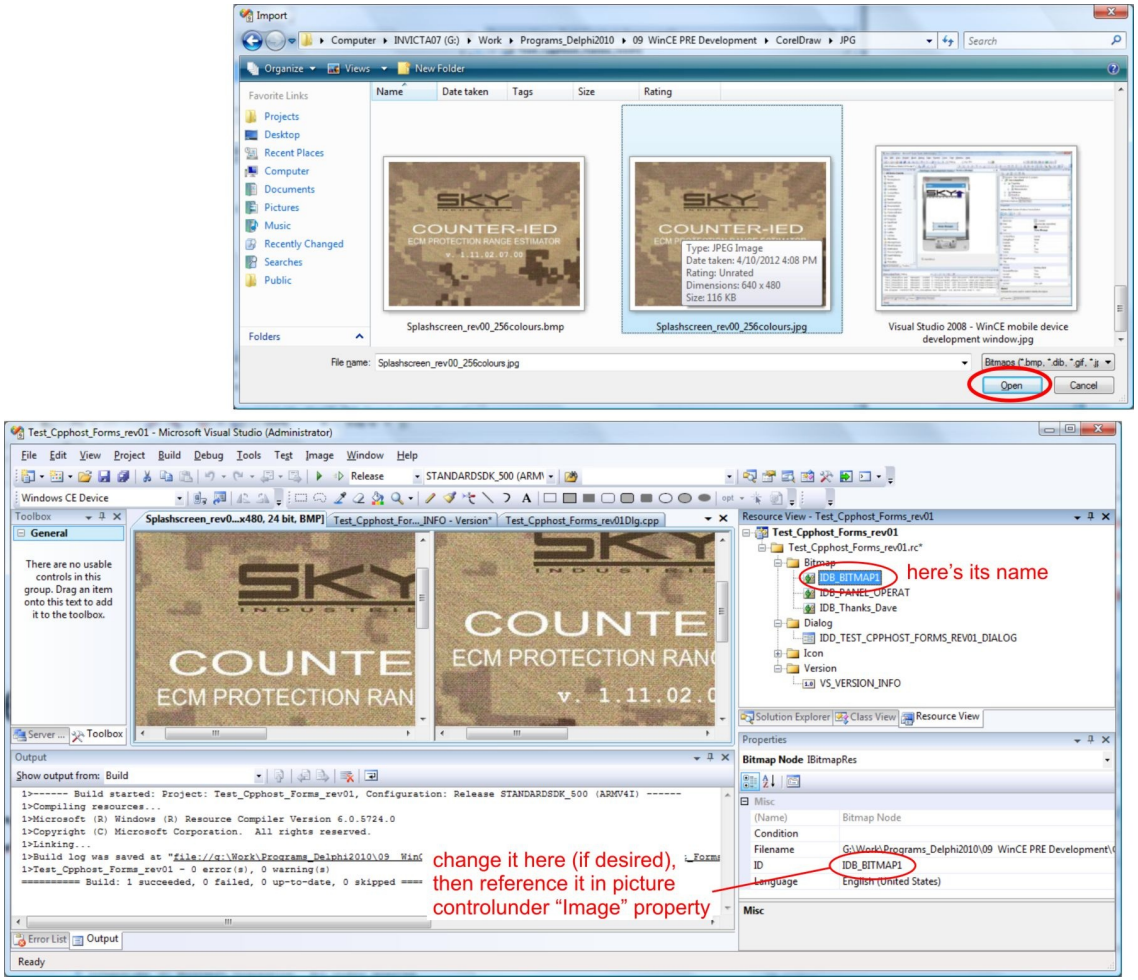


Figure 16: How to add a .BMP file to Visual Studio 2008 Win32/MFC "Picture Control" (cont'd)

Build the project, and copy the executable to the target machine (in my case, a spectrum analyzer). It will run, as long as you picked the right SDK way back when (fig. 17).

Figure 17: Yep, it works

Gigantic Conversion of Delphi to PRISM:

My main objective in all this is to convert a large (~130K lines) chunk of Delphi code so it will compile in PRISM and run on a spectrum

analyzer (SA for short, a piece of test equipment that in my case happens to run WinCE). BUT: I want to keep the Delphi version for development purposes, and whenever I've made enough progress, release another version for the SA. So here's my approach, I don't say it's the right approach or the best approach, but it's what I've done.

I went through my code and found all the instances where file manipulation was happening, that means reading, writing, opening or closing a file. At each place, I used a boolean variable (CompileForDelphi_BOOLEAN) to act as a switch between Delphi version of code and the PRISM version. Write code to duplicate functionality of the Delphi code by writing PRISM code (i.e. using all those .NET things that Delphi doesn't see, but PRISM does, Never mind syntax compatibility for now). If you try to compile the result, all the .NET procedure calls create errors. [BTW I hard-coded the switch variable in the startup procedure TRUE for Delphi compile, FALSE for PRISM compile - or it could be a constant and you change it between TRUE and FALSE depending on the compiler you're using.]

To get rid of the PRISM-syntax errors, create dummy procedures and functions that have the same name as the .NET procedures and functions, but don't put any code in them; they're just empty, dummy procedures to stop the compiler from complaining. Doesn't matter if they're empty because they will *never be called* if the software is compiled using Delphi... because of the switch mentioned above. Read farther and the approach will get clearer.

In the namespace where my gigantic static class is defined (namespace = Delphi unit, same thing, different word, you get used to it), the one that represents all the variables that used to be global, I created two sets of procedures and functions:

SET 1: THE SET FOR DELPHI COMPILER
- it's a bunch of empty procedures and functions that have .NET names, all the ones my PRISM code needs, e.g. the ones that have to do with StreamWriter, StreamReader, etc. Here's what mine look like:

```
TYPE StreamWriter_TYPE = STRING;
TYPE StreamReader_TYPE = STRING;
TYPE TTString = CLASS
PUBLIC
    FUNCTION Join(Separator,String01,String02,String03 : STRING) : STRING;
END;
TYPE TStreamWriter = CLASS
PUBLIC
   PROCEDURE WRITELINE(STRING_LOCAL : STRING);
   PROCEDURE WRITE(STRING_LOCAL : STRING);
   PROCEDURE FLUSH;
```

```
      PROCEDURE CLOSE;
   END;
   TYPE TSeekOrigin = RECORD
      TBEGIN : INTEGER;
   END;
   TYPE TBaseStream = CLASS
     Position : INTEGER;
     Dummy02  : TSeekOrigin;
     PROCEDURE Seek(Dummy01 : INTEGER; Dummy02 : INTEGER);
   END;
   TYPE TStreamReader = CLASS
   PUBLIC
      BaseStream : TBaseStream;
      FUNCTION Peek() : INTEGER;
      PROCEDURE READLINE(VAR STRING_LOCAL : STRING);
      PROCEDURE READ(VAR STRING_LOCAL : STRING);
      PROCEDURE DiscardBufferedData();
      PROCEDURE FLUSH;
      PROCEDURE CLOSE;
   END;
   TYPE TCONVERT = CLASS
   PUBLIC
      FUNCTION TODOUBLE(InputSTRING_LOCAL  : STRING) : DOUBLE;
      FUNCTION TOINT16(InputSTRING_LOCAL   : STRING) : INTEGER;
      FUNCTION TOINT32(InputSTRING_LOCAL   : STRING) : INTEGER;
      FUNCTION TOBOOLEAN(InputSTRING_LOCAL : STRING) : BOOLEAN;
   END;
   PROCEDURE STR_DOUBLEX(Variable_LOCAL   : DOUBLE;  Precision01,Precision02 : INTEGER; VAR
Result_STRING : STRING);
   PROCEDURE STR_INTEGERX(Variable_LOCAL  : INTEGER; Precision01             : INTEGER; VAR
Result_STRING : STRING);
   FUNCTION INT(Variable_LOCAL : DOUBLE)  : INTEGER;
   FUNCTION FRAC(Variable_LOCAL : DOUBLE) : DOUBLE;
   FUNCTION DateTimeToStr(CompileForDelphi_BOOLEAN_LOCAL : BOOLEAN) : STRING;
   FUNCTION Exists(FileNameAndPath_LOCAL  : STRING) : BOOLEAN;

IMPLEMENTATION
```

```
FUNCTION TTString.Join(Separator,String01,String02,String03 : STRING) : STRING;
begin
end;
PROCEDURE TStreamWriter.WRITELINE(STRING_LOCAL : STRING);
begin
end;
PROCEDURE TStreamWriter.WRITE(STRING_LOCAL : STRING);
begin
end;
PROCEDURE TStreamWriter.FLUSH;
begin
end;
PROCEDURE TStreamWriter.CLOSE;
begin
end;
FUNCTION TStreamReader.Peek() : INTEGER;
begin
end;
PROCEDURE TStreamReader.READLINE(VAR STRING_LOCAL : STRING);
begin
end;
PROCEDURE TStreamReader.READ(VAR STRING_LOCAL : STRING);
begin
end;
PROCEDURE TStreamReader.FLUSH;
begin
end;
PROCEDURE TStreamReader.CLOSE;
begin
end;
PROCEDURE TBaseStream.Seek(Dummy01 : INTEGER; Dummy02 : INTEGER);
begin
end;
PROCEDURE TStreamReader.DiscardBufferedData();
begin
end;
```

```
FUNCTION TCONVERT.TODOUBLE(InputSTRING_LOCAL : STRING) : DOUBLE;
begin
end;
FUNCTION TCONVERT.TOINT16(InputSTRING_LOCAL : STRING) : INTEGER;
begin
end;
FUNCTION TCONVERT.TOINT32(InputSTRING_LOCAL : STRING) : INTEGER;
begin
end;
FUNCTION TCONVERT.TOBOOLEAN(InputSTRING_LOCAL : STRING) : BOOLEAN;
begin
end;
{proxy to convert a DOUBLE to STRING, since STR() doesn't exist in PRISM (every variable has a
.ToString method attached to it).  why define STR_DOUBLEX()? because we can define it as a Delphi
STR() function when compiling in Delphi, and as an empty procedure when compiling in PRISM, that
way the Delphi syntax STR() won't cause heartburn for the PRISM compiler because... it won't be
there.}
PROCEDURE STR_DOUBLEX(Variable_LOCAL : DOUBLE; Precision01,Precision02 : INTEGER;VAR Result_STRING :
STRING);
BEGIN
   SYSTEM.STR(Variable_LOCAL:Precision01:Precision02,Result_STRING);
END;
{same thing as above to convert INTEGER to STRING}
PROCEDURE STR_INTEGERX(Variable_LOCAL : INTEGER; Precision01 : INTEGER;VAR Result_STRING : STRING);
BEGIN
   SYSTEM.STR(Variable_LOCAL:Precision01,Result_STRING);
END;
{same as above}
{proxy to take integer part of DOUBLE, since INT doesn't exist in PRISM}
FUNCTION INT(Variable_LOCAL : DOUBLE) : INTEGER;
BEGIN
   Result := ROUND(SYSTEM.INT(Variable_LOCAL));
END;
{same as above}
FUNCTION FRAC(Variable_LOCAL : DOUBLE) : DOUBLE;
BEGIN
   Result := SYSTEM.FRAC(Variable_LOCAL);
```

```
END;
{same as above}
FUNCTION DateTimeToStr(CompileForDelphi_BOOLEAN_LOCAL : BOOLEAN) : STRING;
BEGIN
   Result := SYSUTILS.DateTimeToStr(SysUtils.Now)
END;
{same as above}
FUNCTION Exists(FileNameAndPath_LOCAL : STRING) : BOOLEAN;
BEGIN
   Result := TRUE;{dummy result, the code where this is used is inconsequential in my application}
END;
```

SET 2: THE SET FOR THE PRISM COMPILER
- it's a bunch of empty procedures and functions that have Delphi names, like ASSIGNFILE, WRITELN, WRITE, READLN, ROUND, FRAC, INT, FLUSH, CLOSE, etc. Here's what mine look like:

```
PROCEDURE ASSIGNFILE(ProxyFileName_LOCAL : STRING; PathAndFileName_LOCAL : STRING);
BEGIN
END;
PROCEDURE ASSIGN(ProxyFileName_LOCAL : STRING; PathAndFileName_LOCAL : STRING);
BEGIN
END;
PROCEDURE REWRITE(ProxyFileName_LOCAL : STRING);
BEGIN
END;
PROCEDURE RESET(ProxyFileName_LOCAL : STRING);
BEGIN
END;
PROCEDURE WRITELN(ProxyFileName_LOCAL : STRING; MeaninglessStringVariable_STRING : STRING);
BEGIN
END;
PROCEDURE WRITE(ProxyFileName_LOCAL : STRING; MeaninglessStringVariable_STRING : STRING);
BEGIN
END;
PROCEDURE READLN(ProxyFileName_LOCAL : STRING; MeaninglessStringVariable_STRING : STRING);
BEGIN
```

```
END;
PROCEDURE READ(ProxyFileName_LOCAL : STRING; MeaninglessStringVariable_STRING : STRING);
BEGIN
END;
PROCEDURE FLUSH(ProxyFileName_LOCAL : STRING);
BEGIN
END;
PROCEDURE CLOSE(ProxyFileName_LOCAL : STRING);
BEGIN
END;
PROCEDURE CLOSEFILE(ProxyFileName_LOCAL : STRING);
BEGIN
END;
PROCEDURE STR_DOUBLEX(Variable_LOCAL : DOUBLE; Precision01,Precision02 : INTEGER;VAR Result_STRING :
STRING);
BEGIN
END;
PROCEDURE STR_INTEGERX(Variable_LOCAL : INTEGER; Precision01 : INTEGER;VAR Result_STRING : STRING);
BEGIN
END;
FUNCTION INT(Variable_LOCAL : DOUBLE) : INTEGER;
BEGIN
END;
FUNCTION FRAC(Variable_LOCAL : DOUBLE) : DOUBLE;
BEGIN
END;
FUNCTION DateTimeToStr(CompileForDelphi_BOOLEAN_LOCAL : BOOLEAN) : STRING;
BEGIN
END;
FUNCTION Exists(FileNameAndPath_LOCAL : STRING) : BOOLEAN;
BEGIN
END;
```

To compile for Delphi, I comment out the set that has Delphi names (because these are defined in Delphi, those are the ones the software will use, not the empty ones) and uncomment the PRISM set (they're empty, and they're never called anyway since the switch CompileForDelphi_BOOLEAN is hard coded to TRUE to deselect this code when the software is compiled using Delphi; they're only there to

stop compiler errors).

To compile for PRISM, I do the reverse; I comment out the PRISM set, and uncomment the Delphi set. Again, the empty Delphi procedures are only there to stop PRISM compiler errors; the procedures are never called during execution, because the switch is set to PRISM (CompileForDelphi_BOOLEAN is hard coded to FALSE for PRISM compile).

There are a few other details (which the PRISM compiler will show you and which are covered in the Delphi <–> PRISM “equivalents”), but probably you get the idea.