Document :: The Registry in C++ Builder
Author :: Alan Mills
From "The Bits" Website
http://www.cbuilder.dthomas.co.uk
emails to : jon.jenkinson@mcmail.com

Saving A Form's Positions and Size and Restoring The Details When Next Run

This document is designed as a tutorial to show you how to save a form's position and size. It will then go on to explain how to retrieve these values next time the application is run so that the form starts in the same place on the screen and with the same size as when you quit the application last.

Basically, this tutorial shows you how to save and restore information inside Windows 3.x INI files and inside the System registry. I have simply used the forms size and position as a typical example of what you might use either an INI files of the System registry for.

During the tutorial we will also discuss, where appropriate, how the techniques shown can be used to save and restore application level options as well.

This tutorial is aimed at the beginner but some knowledge of Borland C++ Builder and Windows is assumed.

Colour Coding:

```
Information displayed in this colour is information provided by my copy of BCB.
Information displayed in this colour is information you need to type.
Information displayed in this colour is information which is of general interest.
```

****

Underlying Principals

Many Windows applications on the market today are configurable in that the next time you run an application it 'remembers' how it looked the last time it was run and, on start-up, configures itself to look the same as how you left it the last time. In fact this is one of the little gripes I have with C++ Builder. It doesn't remember the look and feel of how I left it the last time. I always have to resize the property editor to make it usable!!!

This tutorial shows you how to achieve the same thing in your own applications.

How To Save and Restore a Form's Positions and Size Between Runs

Obviously the values we wish to save between runs (position and size) need to be stored somewhere. You could read and write these details from and to a bespoke data file for you application but as the Windows environment already offers us two excellent places designed to store just this type of data, why bother. I shall not consider any bespoke data files in this tutorial but will explain how to use the in-built Windows devices designed for the purpose.

If you haven't guessed already, the two devices I am referring to are Windows INI files and the Windows System Registry.

Borland C++ Builder, indeed, offers us two ways of reading and writing to the System Registry so that gives us three methods of saving and restoring application properties. We shall discuss each in turn.

Each method discussed involves creating a new application equivalent to each of the others. I have used the same names in each application and I store each app in a separate directory. You should do the same.

Method 1, using Windows 3.x INI files is supposed to be redundant in Windows 95 and NT as we are all now supposed to be using the System Registry. I therefore recommend that you use Method 3, below, in any new applications you write as this is the true System Registry method. Methods 1 and 2 are included here for completeness and for INI files which you want to port up.

Step One : Create an application

Method 1 - Windows INI Files

Step One : Create the Application

a) create a new directory for the application. It doesn't matter what the directory is called.
b) From within C++ Builder, create a new app using the menu File|New Application
c) Add a TBitBtn component to the form and change its 'kind' property to become a 'Close' button.

The reason for this is to show that the values we wish to save to the ini file and to the registry will work when we close down the app from using both a Close button and from using the close box in the system menu (the little box with a cross in it at the top right hand corner of the window).

d) Save the application to the directory you just created. For this tutorial I have assumed the files keep their default names of 'project1' and 'unit1'.

e) Compile and run the application. Resize the form and drag it to a new position on the screen then close the form using either the button of the close option from the system menu.
f) Re-run the app. Notice how it appears just as it did on start-up the previous time and not how it was when it was last closed.

Step Two : Saving Application Properties to the INI File

a) Create an OnClose Event for the form (Click on the form, go to the properties editor, Click on the Events Tab and double click on the OnClose event). This will give you an empty function which looks like

```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
}
```

b) Add the following header file to the top of the unit

```
#include <vcl\inifiles.hpp>
```

This line ensures that the compiler knows about the TIniFile component.

c) Add the following line of code to the function.

```
TIniFile *WinIni = new TIniFile("IniTester.ini");
```

This line does three things.

1. It defines a pointer to a TIniFile object called WinIni,
2. Creates a new TIniFile object in memory and links the pointer to it,
3. Gives the TIniFile object the name of the INI file to create/open ("IniTester.ini" in this case - you can choose any name you like). The filename chosen will be created in you Windows directory (typically C:\WINDOWS). If you wish you could supply a complete pathname instead of just a filename to place the INI file exactly where you want it, and finally
4. Opens the INI file for use.

d) Add the code which writes the application properties to the INI file just opened. These code lines should go below the code above.

```
WinIni->WriteInteger("Positions", "FormTop", Form1->Top);
WinIni->WriteInteger("Positions", "FormLeft", Form1->Left);
WinIni->WriteInteger("Size", "FormHeight", Form1->Height);
WinIni->WriteInteger("Size", "FormWidth", Form1->Width);
```

To store the form's position and size we need four properties from the form itself (Top and Left to specify position, and Height and Width to specify size).

As all our values we wish to store are integers we use the WriteInteger method of the TIniFile object created. The parameters to this are as follows:

1. The section, within the INI file, to (create and) write the value to,
2. The identifier for the stored value, and
3. The value itself. Here we take the values straight from the Form object itself. If you are storing application options, these might be from variables or from properties of other components.

e) Add the code to remove the TIniFile object from memory now we are finished with it.

```
delete WinIni;
```

We could just as easily call the Free method of the TIniFile object with the statement

```
WinIni->Free();
```

to do the same job. You can choose which method you like best!

f) Save, compile and run the app.

g) Go to you windows directory and locate the file IniTester.ini (or whatever you chose as your INI filename). It should exist and if you view it (possibly in Notepad) you will see the two section 'Positions' and 'Size', each with two values associated with it. You've now created an INI file so smile :)

Step Three : Reading App Properties From the INI File

This task is remarkably similar to writing to the INI file (Step Two) as you'll see.

a) Create and OnCreate Event for the form (Click on the form, got to the properties editor, Click on the Events Tab and double click on the OnCreate event). This will give you an empty function which looks like

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
}
```

b) Add the following line of code to the function.

```
TIniFile *WinIni = new TIniFile("IniTester.ini");
```

This is the same as in step 2 so you may wish to just cut and paste it.

c) Add the code which reads the application properties to the INI file just opened and set the appropriate form properties. These code lines should go below the code above.

```
Form1->Top = WinIni->ReadInteger("Positions", "FormTop", Form1->Top);
Form1->Left = WinIni->ReadInteger("Positions", "FormLeft", Form1->Left);
Form1->Height = WinIni->ReadInteger("Size", "FormHeight", Form1->Height);
Form1->Width = WinIni->ReadInteger("Size", "FormWidth", Form1->Width);
```

As all our values we wish to store are integers we use the ReadInteger method of the TIniFile object created. The parameters to this are as follows:

1. The section, within the INI file, to read the value from,
2. The identifier for the stored value, and
3. The default value to return if the INI file, section or value identifier cannot be read (or doesn't exist yet!). We can simply default to the forms values as defined in the C++ Builder designer.

Note that we read a value from the INI file and set the appropriate form property in a single statement.

d) Add the code to remove the TIniFile object from memory now we are finished with it.

```
delete WinIni;
```

Again, we could just as easily call the Free method of the TIniFile object with the statement

```
WinIni->Free();
```

to do the same job. It's up to you!

e) Add code to redraw the form now we have changed its values.

```
Form1->Refresh();
```

Step Four : All done!

a) Save the application
b) Compile the application
c) Run the app
d) Resize form and reposition form then close down the app
e) Re-run the app.
f) Notice the form is in the same position and of the same size as when you last close down the app and smile broadly :))

Full Code

Here is the full source code from file unit1.cpp as it appears on my system. You can use it to cut and paste the code you want and to compare your example with mine.

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#include <vcl\inifiles.hpp>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
  TIniFile *WinIni = new TIniFile("IniTester.ini");

  WinIni->WriteInteger("Positions", "FormTop", Form1->Top);
  WinIni->WriteInteger("Positions", "FormLeft", Form1->Left);
  WinIni->WriteInteger("Size", "FormHeight", Form1->Height);
  WinIni->WriteInteger("Size", "FormWidth", Form1->Width);

  delete WinIni;

  return;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  TIniFile *WinIni = new TIniFile("IniTester.ini");

  Form1->Top = WinIni->ReadInteger("Positions", "FormTop", Form1->Top);
```

```
  Form1->Left = WinIni->ReadInteger("Positions", "FormLeft", Form1->Left);
  Form1->Height = WinIni->ReadInteger("Size", "FormHeight", Form1->Height);
  Form1->Width = WinIni->ReadInteger("Size", "FormWidth", Form1->Width);

  delete WinIni;
  Form1->Refresh();

  return;
}
//---------------------------------------------------------------------------
```

Method 2 - Windows System Registry I

Assuming you've gone through Method 1, above, then this is so easy it's unbelievable. C++ Builder gives us a lovely component in the VCL called TRegIniFile which reads and writes to the System Registry but uses properties and methods similar to the TIniFile component used above. It's designed to ease the pain of porting existing Windows 3.x C++ Builder programs (I'm not too convinced about this statement as C++ Builder only generates 32 bit code to my knowledge so why are it's programs being run on Windows 3.x??) to a Windows 95 or NT environment.

Step One : Create an app

Same as Step One for Method 1, above.

Step Two : Adding the code

I'm going to do everything in one step here, in Method 2, because we have already done the hard work in Method 1.

a) Create your app exactly as you did for Method 1 with exactly the same code except for the following lines.

1. Use header file

```
#include <vcl\registry.hpp>
```

instead of

```
#include <vcl\inifiles.hpp>
```

2. In both functions replace the line

```
TIniFile *WinIni = new TIniFile("IniTester.ini");
```

with

```
TRegIniFile *WinIni = new TRegIniFile("IniTester\\TRegIniFile");
```

This creates an object of type TRegIniFile rather than TIniFile.

The parameter you pass to this constructor is the name of the key you wish to create in the System Registry. I have used a two level key as I will keep the upper level the same for Method 3 and just change the lower level.

Step Three : All done!

Easy wasn't it?

a) Save the application
b) Compile the application
c) Run the app
d) Resize form and reposition form then close down the app
e) Re-run the app.
f) Notice the form is in the same position and of the same size as when you last close down the app and smile broadly :))

Full Code

I don't quote the full code for Method 2 because, as I explained earlier, it is so similar to Method 1. Only three lines are different!

Method 3 - Windows System Registry II

This is the method by which it is recommended that all new applications adhere to. The System Registry has been introduced into Windows95 and NT for exactly the type of thing we are trying to implement. INI files from Windows 3.x are considered outdated now. Although Method 2 reads and writes to the System Registry it does so using a feature of C++ Builder designed to ease porting of applications from Window 3.x to Windows 95 and NT. This last method, Method 3, uses the TRegistry component of C++ Builder and so should be considered the 'normal expected' method of accessing the System Registry.

The example application built in this section is, as explained earlier, equivalent to the two example applications above.

Step One : Create the Application

Same as Step 1 for Methods 1 and 2, above.

Step Two : Saving application properties to the System Registry

a) Create an OnClose event for the form (in the usual manner). This will give an empty function looking like this:

```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
}
```

b) As with Method 2, add the header

```
#include <vcl\registry.hpp>
```

to the top of the unit source code. This lets the compiler know all about the TRegistry component we will be using.

c) Add the code to define the instance of the TRegistry component we will use and point it at the root key of the System Registry that we require.

```
TRegistry *MyReg;     //Define pointer
```

```
MyReg = new TRegistry;   //Allocate space and assign pointer

MyReg->RootKey = HKEY_CURRENT_USER;    //Set Root key inside registry
```

d) Add code to open up, for use, the specific keys we require in the System Registry.

```
//Write position info
if (MyReg->OpenKey("\\IniTester\\TRegistry\\Positions", TRUE))
{
       //Code to write position info goes here.
}


//Write size info
if (MyReg->OpenKey("\\IniTester\\TRegistry\\Size", TRUE))
{
       //Code to write size info goes here.
}
```

To achieve a similar result to that obtained from Method 2 we have to write the position info and the size info to two slightly different key paths in the System Registry. To do this we have to perform two opens using the OpenKey() method of the TRegistry component. Only if each open succeeds do we perform the necessary writes. Opening the keys for reading (discussed later) works in the same fashion.

The first parameter to the OpenKey method is the key path to open. Take the positions path. The actual path we require, from the root key id actually '\iniTester\TRegistry\Positions'. Remember that in 'C' strings the '\' character acts as an escape and is used to include special character in string constants like a newline or a tab. To actually include a backslash character ('\') we have to place two together.

Also, be careful to note that the string starts with '\\'. Starting the key name with '\\' ensures that the path is relative to the root key. Excluding it will open a key with the same name but relative to the current key opened in the System Registry. This may, on occasion, be the same thing but DO NOT rely on it.

The second, and last, parameter to the OpenKey method is a Boolean that specifies whether we the call should create the key path if it does not already exist. Given that on the first run of our program, the path does not exist (it would be a bit of a coincidence if it did!) then we need to make sure we create the key path when we first try to write to it. We therefore set the flag to TRUE.

e) Add the code to write each of the application properties. We have four to add. The code for each is of a similar format with properties Form1->Top and Form1->Left to be added inside the OpenKey if statement for key 'Positions' and the properties Form1->Width and Form1->Height to be added into the OpenKey if statement for key 'Size' as follows

```
//Write position info
if (MyReg->OpenKey("\\IniTester\\TRegistry\\Positions", TRUE))
{
      //Write TOP
      try
      {
        MyReg->WriteInteger("FormTop", Form1->Top);
      }
      catch (...)
      {
        ;  //do nothing
      }


      //Write LEFT
      try
      {
```

```
        MyReg->WriteInteger("FormLeft", Form1->Left);
      }
      catch (...)
      {
            ;  //do nothing
      }
   }


//Write size info
if (MyReg->OpenKey("\\IniTester\\TRegistry\\Size", TRUE))
{
     //Write WIDTH
     try
     {
       MyReg->WriteInteger("FormWidth", Form1->Width);
     }
     catch (...)
     {
           ;  //do nothing
     }


     //Write HEIGHT
     try
     {
       MyReg->WriteInteger("FormHeight", Form1->Height);
     }
     catch (...)
     {
           ;  //do nothing
     }
   }
```

Notice that each use of the WriteInteger method of the TRegistry component is wrapped inside a try ... catch construct which makes the code look long winded. This is quite necessary. Unlike for TRegIniFile and TIniFile (at least it's not documented in my help files!) the WriteInteger method of TRegistry raises an exception if the write cannot be made. We must make sure we trap the exception if it occurs regardless of whether we wish to take any special action for it. In this example we do not want to take any special action. if we couldn't write the values then there's nothing we can do about it until we come to try and read the values next time so the catch statement simply does nothing.

There are two parameters to the WriteInteger method. The first specifies the identifier for the value and the second parameter specified the value to store there.

We could have treated the writing of these values in pair and put both writes for each key inside a single try ... catch statement but for completeness I've treated them individually.

We will notice that reading values is very similar to this.

f) Finally, just as in the previous methods we should free up the instance of the TRegistry component used so add the following line to the bottom of the function.

```
MyReg->Free();
```

g) Save you application and compile if you wish.
At this stage if you were to run your application and quite. You should be able to see the registry entries created using the Registry Editor.

Step Three : Reading application properties from the System Registry

Just as with the previous methods, the code for reading values back is remarkably similar in format to that of writing the values.

a) Create an OnCreate event handler for the form in the usual manner. The empty function should look as follows in the code window:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
}
```

b) This is the same as (c) of Step Two, above. Add the code to define the instance of the TRegistry component we will use and point it at the root key of the System Registry that we require.

```
TRegistry *MyReg;     //Define pointer

MyReg = new TRegistry;   //Allocate space and assign pointer

MyReg->RootKey = HKEY_CURRENT_USER;    //Set Root key inside registry
```

c) This is the same as (d) in Step Two with the exception that the second parameter to the call to the OpenKey method is FALSE to indicate NOT to create the key path if it doesn't exists. After all, we're only reading!

Add code to open up, for use, the specific keys we require in the System Registry.

```
//Write position info
if (MyReg->OpenKey("\\IniTester\\TRegistry\\Positions", FALSE))
{
       //Code to write position info goes here.
}


//Write size info
if (MyReg->OpenKey("\\IniTester\\TRegistry\\Size", FALSE))
{
       //Code to write size info goes here.
}
```

d) Add code to perform the read and assign to appropriate form property.

```
//Get position info
if (MyReg->OpenKey("\\IniTester\\TRegistry\\Positions", FALSE))
{
       //Get TOP
       try
              {
              Form1->Top = MyReg->ReadInteger("FormTop");
              }
              catch (...)
              {
                     ; // Do nothing.  Leave Form->Top as it is
              }

              //Get Left
       try
              {
              Form1->Left = MyReg->ReadInteger("FormLeft");
              }
              catch (...)
              {
                     ; // Do nothing.  Leave Form->Top as it is
              }
}
```

```
//Get size info
if (MyReg->OpenKey("\\IniTester\\TRegistry\\Size", FALSE))
{
        //Get WIDTH
        try
                {
                Form1->Width = MyReg->ReadInteger("FormWidth");
                }
                catch (...)
                {
                        ; // Do nothing.  Leave Form->Top as it is
                }

                //Get HEIGHT
        try
                {
                Form1->Height = MyReg->ReadInteger("FormHeight");
                }
                catch (...)
                {
                        ; // Do nothing.  Leave Form->Top as it is
                }
}
```

Note that, again, if we receive an exception from any of the read then we do nothing. For form properties such as these then a good default for the values is the values currently held for the form from the C++ Builder designer. If we were reading in application options then our catch statement should set some sort of default value.

e) Finally, just as in the previous methods we should free up the instance of the TRegistry component used so add the following line to the bottom of the function.

```
MyReg->Free();
```

and then refresh the display of the form now that we have modified its properties so add the line

```
Form1->Refresh();
```

f) Save the application and compile it. Now that we are both reading and writing our values from and to the System Registry executing the program should result in the form being displayed, on start-up, in the same place and with the same size as it was when the application was last closed.

e) Smile VERY broadly **:))**

Full Code.

For completeness, here is a copy of the full unit source code as I have it on my machine. Please use it to cut and paste the non C++ Builder provided code to speed up following this tutorial.

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#include <vcl\registry.hpp>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
```

```
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
  TRegistry *MyReg;     //Define pointer

  MyReg = new TRegistry;   //Allocate space and assign pointer

  MyReg->RootKey = HKEY_CURRENT_USER;    //Set Root key inside registry

  //Write position info
  if (MyReg->OpenKey("\\IniTester\\TRegistry\\Positions", TRUE))
  {
        //Write TOP
    try
    {
      MyReg->WriteInteger("FormTop", Form1->Top);
    }
    catch (...)
    {
        ;  //do nothing
    }


        //Write LEFT
    try
    {
      MyReg->WriteInteger("FormLeft", Form1->Left);
    }
    catch (...)
    {
      ;  //do nothing
    }
  }


      //Write size info
      if (MyReg->OpenKey("\\IniTester\\TRegistry\\Size", TRUE))
      {
        //Write WIDTH
        try
        {
          MyReg->WriteInteger("FormWidth", Form1->Width);
        }
        catch (...)
        {
             ;  //do nothing
        }


        //Write HEIGHT
        try
        {
          MyReg->WriteInteger("FormHeight", Form1->Height);
        }
        catch (...)
        {
            ;  //do nothing
        }
      }

  MyReg->Free();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  TRegistry *MyReg;     //Define pointer

  MyReg = new TRegistry;   //Allocate space and assign pointer

  MyReg->RootKey = HKEY_CURRENT_USER;    //Set Root key inside registry

  //Get position info
  if (MyReg->OpenKey("\\IniTester\\TRegistry\\Positions", FALSE))
  {
```

```
     //Get TOP
    try
    {
      Form1->Top = MyReg->ReadInteger("FormTop");
    }
    catch (...)
    {
        ; // Do nothing.  Leave Form->Top as it is
    }

            //Get Left
    try
    {
      Form1->Left = MyReg->ReadInteger("FormLeft");
    }
    catch (...)
    {
              ; // Do nothing.  Leave Form->Top as it is
    }
  }


  //Get size info
  if (MyReg->OpenKey("\\IniTester\\TRegistry\\Size", FALSE))
  {
        //Get WIDTH
        try
            {
          Form1->Width = MyReg->ReadInteger("FormWidth");
            }
            catch (...)
            {
              ; // Do nothing.  Leave Form->Top as it is
            }

            //Get HEIGHT
        try
            {
          Form1->Height = MyReg->ReadInteger("FormHeight");
            }
            catch (...)
            {
              ; // Do nothing.  Leave Form->Top as it is
            }
      }

      MyReg->Free();
      Form1->Refresh();
}
//---------------------------------------------------------------------------
```

****

A Final Note

I hope this gives a reasonable flavour of how you can read and write to INI files and the System Registry yourself. Please email me your comments to ***millsad@usa.net***.

No liability is accepted by the author(s) for anything which may occur whilst following this tutorial

****