Work with the Windows Registry from within Visual Basic 6

In this month's article, I want to show you how easy it is to write to, and read from, the Windows Registry from within Visual Basic. Using the Windows Registry allows your Visual Basic to save a variety of data from one session of your program to the next---such as the locations of files, user preferences---virtually anything that you can think of. This ability can give your program a very polished look and feel, and make the users of your program very happy indeed.

Are there alternatives to using the Windows Registry? For sure. You can elect to save that same information to a disk file or to a database table. But for ease of use, nothing beats the Windows Registry. Just remember that the ideal use for the Windows Registry is small chunks of data---don't try to use it as a Database table. If you have the need to save voluminous amounts of data, don't use the Windows Registry---use either a disk file or a Database table instead.

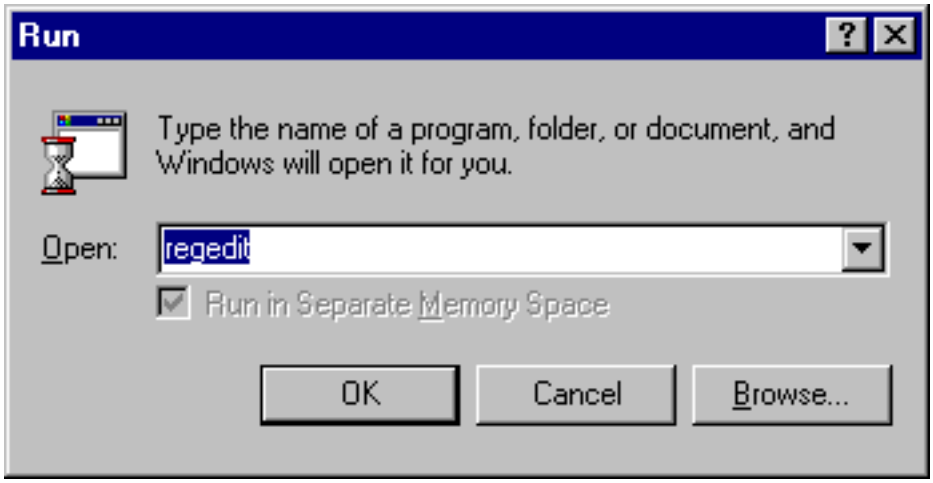
I'm frequently asked by my students if writing to the Windows Registry is dangerous? That is, can you damage the Windows Registry by writing data to it via Visual Basic? The answer is a resounding 'No'. There's no more danger in your Visual Basic program writing information to the Windows Registry than there is in any other Windows program writing to it.

However, the Windows Registry has been known to become corrupted at any time---so I thought the first thing I would do is show you how to back up your Registry before discussing Writing to and Reading from the Registry.

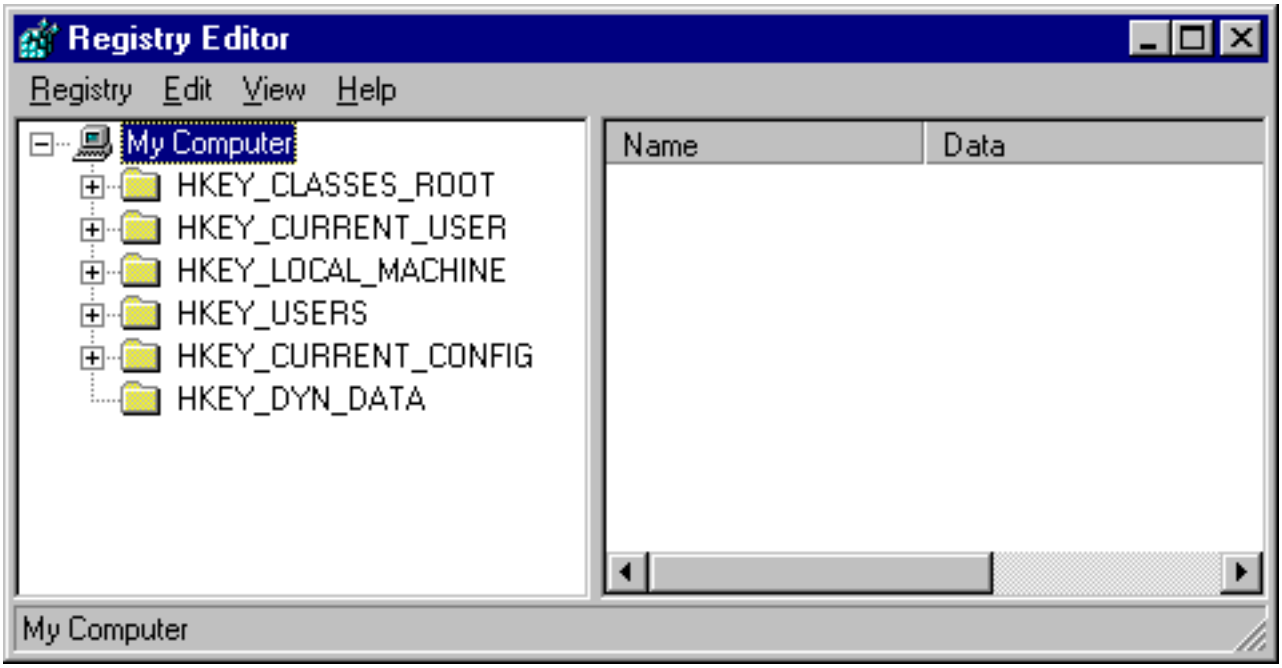
Backing up the Windows Registry

Backing up the Windows Registry is very easy, and results in the generation of a text file. You can save the file to your hard drive or to a floppy (although it may be so large that it won't fit on a floppy without compression). As with other types of backups, it makes sense to save it to a floppy or CD-ROM, and to store it in a location away form your PC.

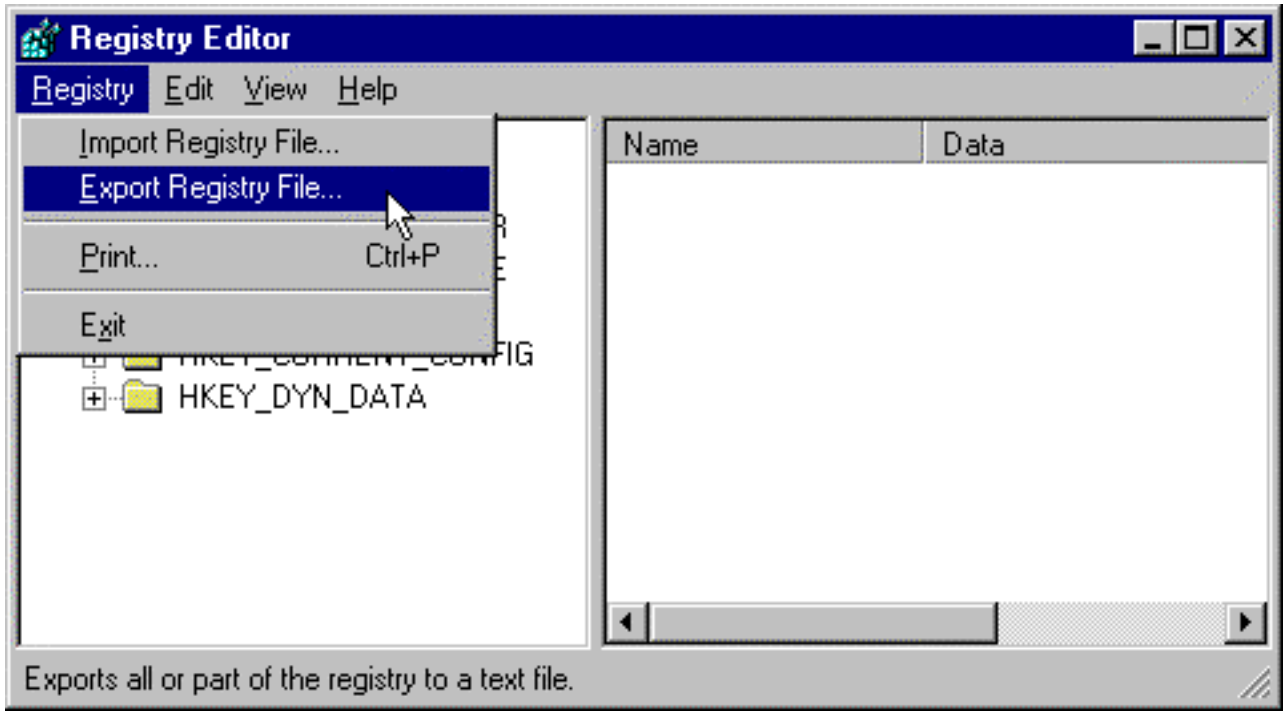
To backup the Registry, click on the Windows Start Button, then click on Run. Type **regedit** into the textbox, and click on the OK button.



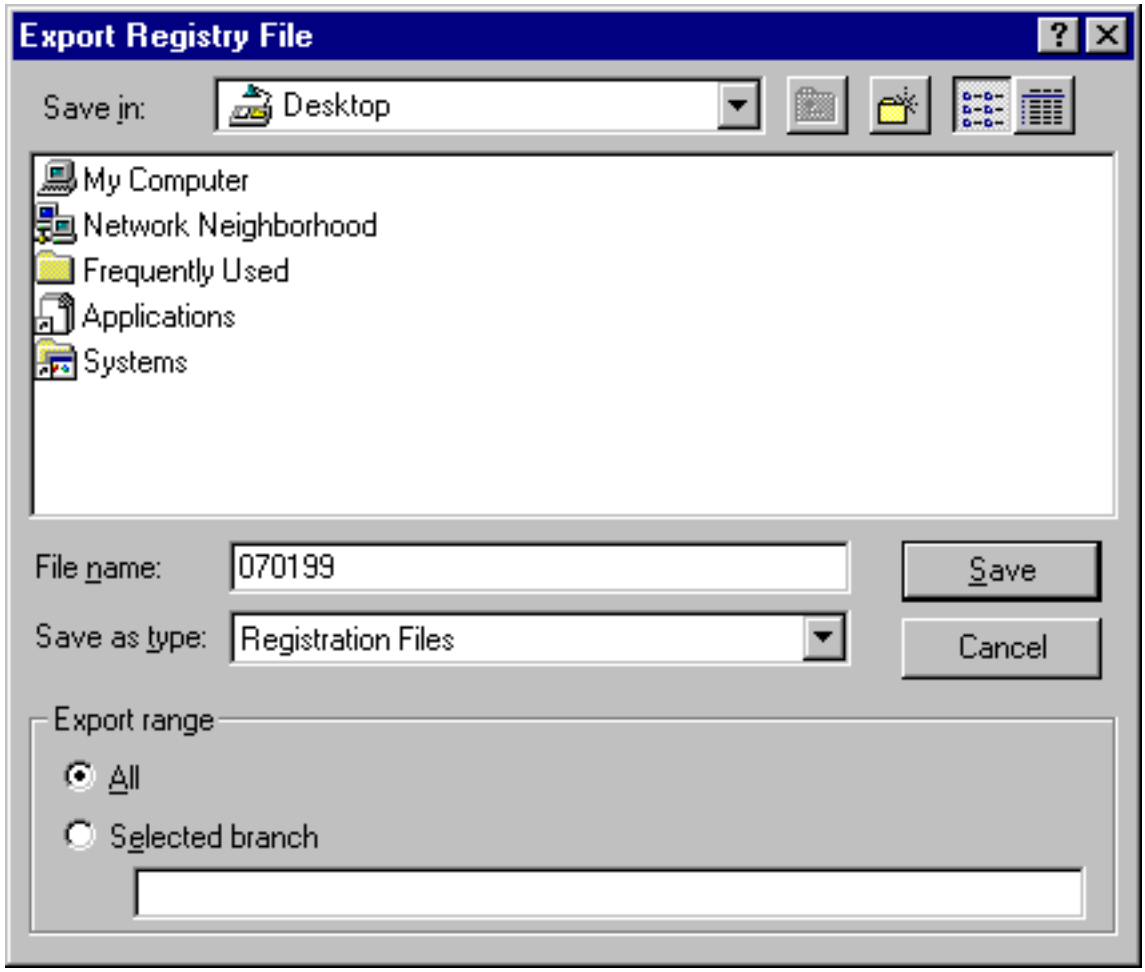
The Windows Registry Editor will appear.



Now Select **Registry-Export Registry File** from the Main Menu.



A dialog box will appear asking you to specify the name for the Registry Backup file. I usually name my Registry Backup files according to the date I backed it up. For instance, since today is July 1, 1999, name the Registry Backup **070199**. Make sure that the Export Range of 'All' is selected. The extension .reg will automatically be added to the file.



Now click on the Save button, and a backup of the Registry will be saved for you. At this point, it would be a good idea to save this file someplace other than your computer's hard drive. If you're at all curious, you can use Notepad to view the contents of the Registry backup---you may be surprised to find that the backup is just a plain old Text file.

Restoring the Registry is not nearly as easy as backing it up, and I'm not going to get into that here. The important thing is that you now have peace of mind that if something does happen to your Registry, there is a way to restore it. Believe me, if anything corrupts your Registry it won't be anything I show you here today.

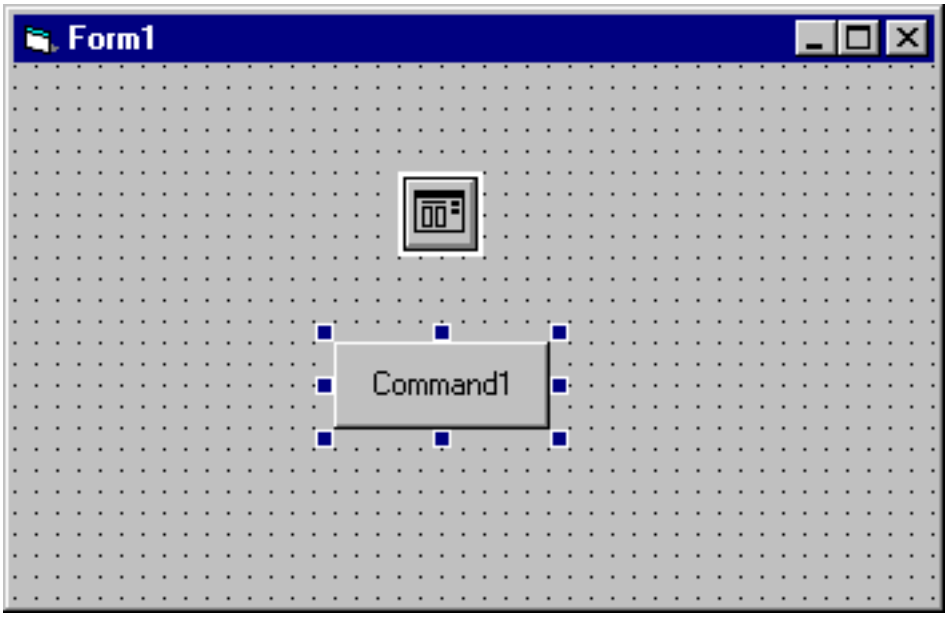
Visual Basic provides four Registry Commands for our use. Let's examine writing to the Registry first.

Writing to The Windows Registry

Throughout my discussion of Visual Basic and the Registry, I'd like to follow a simple example.

I'll create a Visual Basic project with a single form that permits the user to change the BackColor property of the form. I'll then use a Visual Basic statement to save that BackColor property to the Windows Registry so that when the program is started again, the form's BackColor can be changed to the setting in the Windows Registry using the Visual Basic function that reads from the Windows Registry.

Let's start by placing a Common Dialog Control and a Command Button on the form.



In the Click Event Procedure of the Command Button, we'll place code to display the Color Dialog Box of the Common Dialog Control, and to use the Visual Basic SaveSetting statement to store the value of the Color Property of the Common Dialog Control in the Windows

Registry. Here's the format of the SaveSetting Statement.

SaveSetting appname, section, key, setting

The **SaveSetting** statement syntax has these named arguments:

Argument Description

appname Required. String expression containing the name of the application or project to which the setting applies.

section Required. String expression containing the name of the section where the key setting is being saved.

key Required. String expression containing the name of the key setting being saved.

setting Required. Expression containing the value that key is being set to.

I know this syntax looks a little confusing---let me try to de-mystify this for you a bit.

Remember, the Windows Registry is just a database, and a database is very much like a file, which is composed of records and fields. The first three arguments of the SaveSetting statement correspond to fields in a record---in this case, the Registry record. The field names just happen to be appname, section, key and setting, and the syntax for this Visual Basic statement dictates that we supply a value for each field. The great thing about the SaveSetting statement is that you can specify any values you want for appname, section and key. There's no magic to the names you chose for appname, section, and key. They can really be anything you want---but of course, you should pick something meaningful.

The important 'field' is the setting field--the fourth argument---as that will be the value that we are really interested in---in this case, the user's selection of a color for the form. Let me show you the code for the Click Event Procedure of the Command Button---I think that will help to clarify this for you.

```
Private Sub Command1_Click()

CommonDialog1.ShowColor
Form1.BackColor = CommonDialog1.Color
SaveSetting "MyApplication", "STARTUP", "BackColor", CommonDialog1.
Color

End Sub
```

That's all there is to it, really. The first thing we do is display the Color Dialog Box of the Common Dialog Control using the ShowColor method…

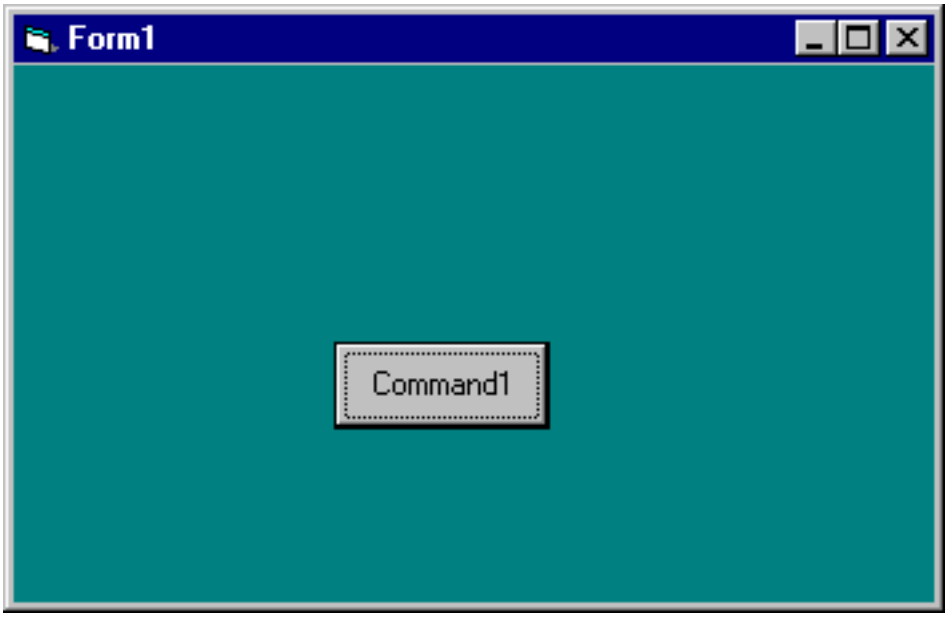
CommonDialog1.ShowColor

At this point, it's just a matter of waiting for the user to make a selection.

When they do, the Long Integer value equating to the color they have selected will be stored in the Color Property of the Common Dialog Control, and this line of code then sets the form's BackColor property equal to that value…

Form1.BackColor = CommonDialog1.Color

That changes the form's BackColor

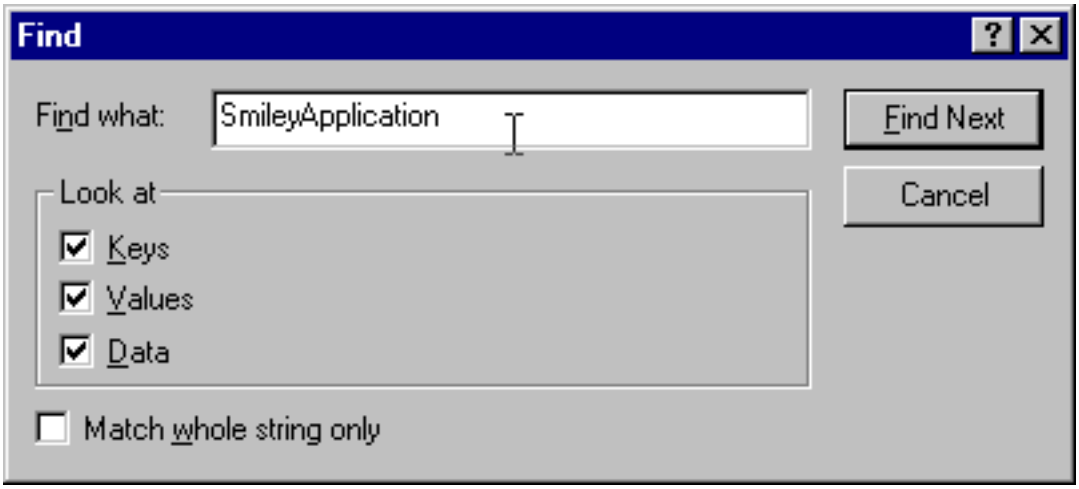


which is fine, except if we don't do anything else here, and then end the program, the BackColor of the form will still be the default color when we next start the program. We take care of that by storing the user's preferred color in the Windows Registry using the Visual Basic SaveSetting statement, like this…

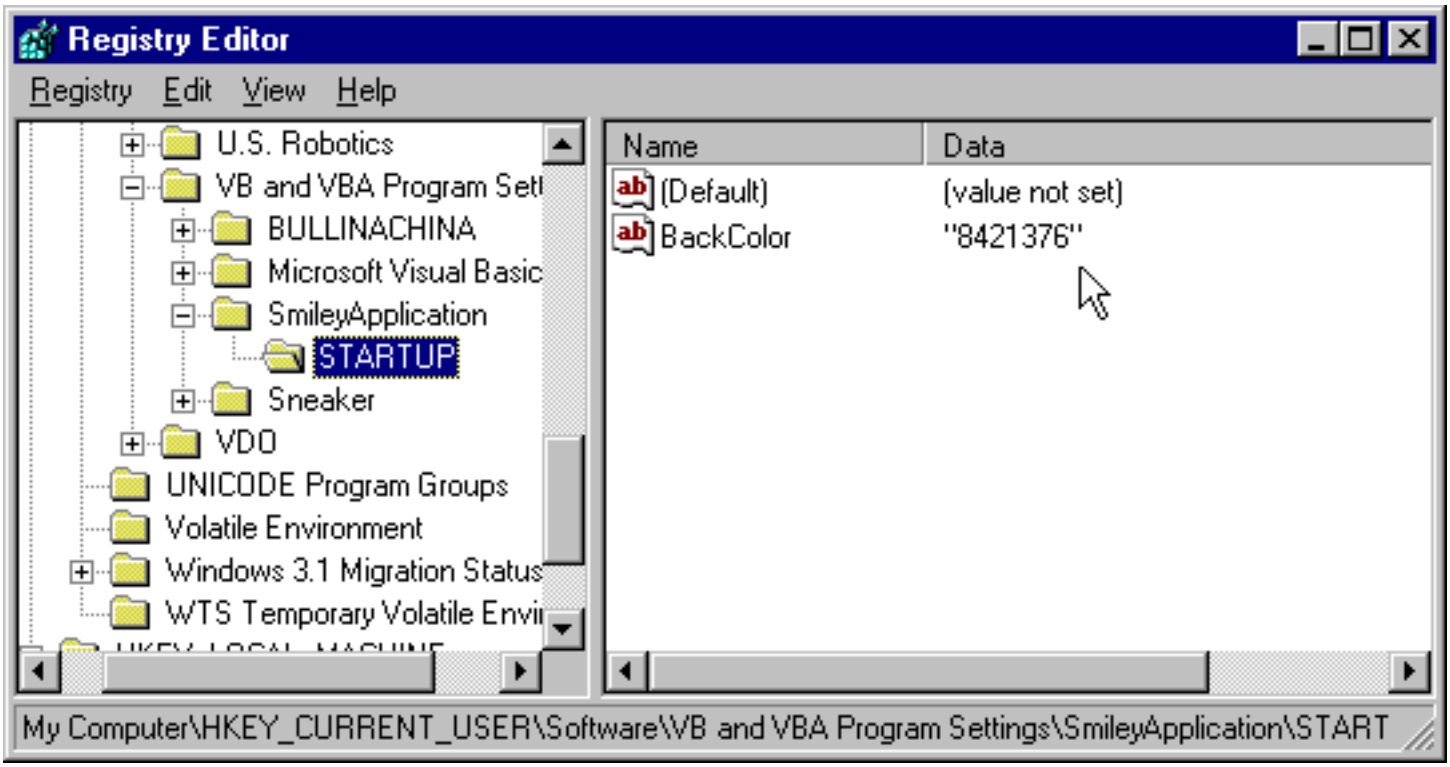
```
SaveSetting "SmileyApplication", "STARTUP", "BackColor",
CommonDialog1.Color
```

Smileyapplication is the AppName value, **STARTUP** is the Section value, **BackColor** is the key value, and the actual value stored in the Registry is the Color Property of the Common Dialog Control. I should mention here that to keep things simple, I haven't included any Error Handling code to deal with the possibility that the user clicks on the Cancel button of the Color Dialog Box. If they do, then this code will result in the form's BackColor being set to black. For that reason, you should set the CancelError property of the Common Dialog Control to True, and code an Error Handler in this event procedure (For more information, check Chapter 14 of my book, Learn to Program with Visual Basic.)

Once this line of code executes, a 'record' is written to the Windows Registry with these four 'field' values. In fact, if you want to check this out for yourself, you can fire up the Registry Editor again, and search for the value by selecting Edit-Find from the Registry Editor's Menu Bar.



You should find an entry for SmileyApplication in the Registry, and if you expand its folder and look in the STARTUP folder, you should find the value for the BackColor key..



Reading from The Windows Registry

Now that we have the user's preferred color saved in the Windows Registry, we need to access the value for that color whenever our program starts up. We'll use the Visual Basic GetSetting function to read this value from the Windows Registry, and a good place to execute that function is in the Load Event Procedure of the form. Before I show you that code, let me show you the format for the GetSetting function.

GetSetting(appname, section, key[, default])

The **GetSetting** function has these named arguments:

Argument Description

appname Required. String expression containing the name of the application or project whose key setting is requested.

section Required. String expression containing the name of the section where the key setting is found.

key Required. String expression containing the name of the key setting to return.

default Optional. Expression containing the value to return if no value is set in the key setting. If omitted, default is assumed to be a zero-length string ("").

As you can see, the first three arguments of the GetSetting function is identical to the SaveSetting statement---which makes sense since these first three arguments really represent the 'key' to the Registry record we write with the SaveSetting function.

The difference lies in the fourth argument.

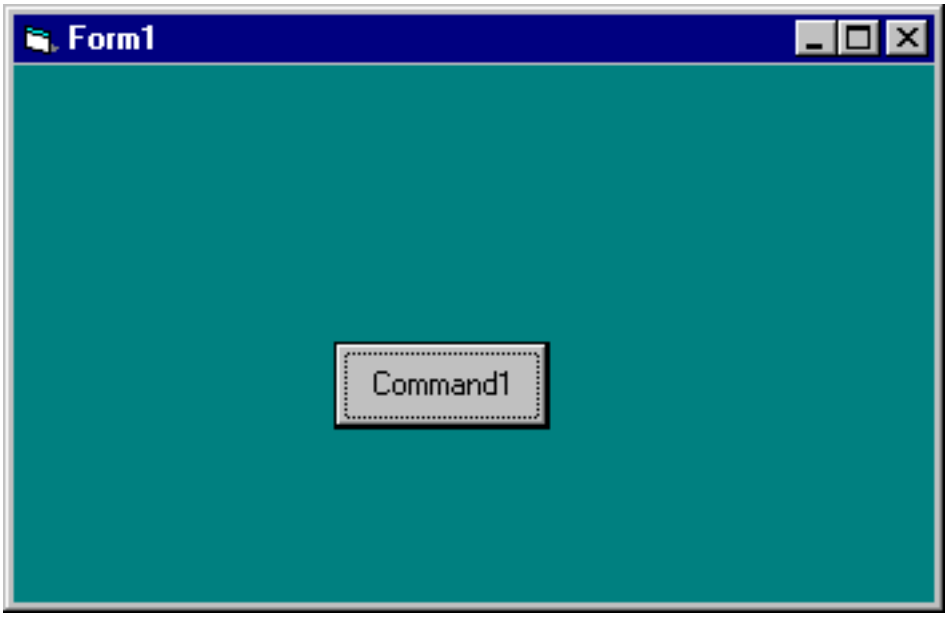
With the **SaveSetting** Statement, the fourth argument was the value to be written to the Registry. For the GetSetting function, the fourth argument is an optional argument used to specify a default value to be returned in the event that the entry specified cannot be located in the Windows Registry. It's always a good idea to specify a default value---if you think about it, it's possible that when the code is executed to read an entry in the Windows Registry, the code that wrote that entry may not have been run yet. Here's the code to place in the Load Event Procedure of the form.

```
Private Sub Form_Load()

Form1.BackColor = GetSetting("SmileyApplication", "STARTUP",
"BackColor", vbWhite)

End Sub
```

As you can see, the first three arguments are identical to the SaveSetting statement---the fourth argument, here specified as the Intrinsic Constant vbWhite, ensures that if the Windows Registry entry cannot be found, that the BackColor of the form is set to white. If we now run this program, the form's BackColor property will be set to the value saved in the Registry---with the result that the form is loaded with the same color the program ended with…



Deleting entries in The Windows Registry

If you're concerned with cluttering the Registry with entries that you create, you can always erase your entries by using the DeleteSetting statement. Here's the format for the DeleteSetting statement:

DeleteSetting *appname*, *section*[, *key*]

The **DeleteSetting** statement has these named arguments:

Argument Description

appname Required. String expression containing the name of the application or project whose key setting is requested.

section Required. String expression containing the name of the section where the key setting is found.

key Optional. String expression containing the name of the key setting to return.

DeleteSetting can be used in one of two ways.

You can use it to delete the appname, section and key entry, similar to the way you created the entry to begin with. Like this:

DeleteSetting "SmileyApplication", "STARTUP", "BackColor"

Or, you can use it to delete ***every*** entry in the Appname and Section, like this:

```
DeleteSetting "SmileyApplication", "STARTUP"
```

GetAllSettings

In the event that we somehow manage to lose track of the entries we have made in the Windows Registry for a particular Appname and Section, and we don't care to use RegEdit to find all of our entries, Visual Basic provides us a function called GetAllSettings which can be used to return the Registry entries for a particular Appname and Setting. Here's the format for the GetAllSettings function:

GetAllSettings(*appname, section*)

The **GetAllSettings** function syntax has these two named arguments:

Argument Description

appname Required. String expression containing the name of the application or project whose key setting is requested.

section Required. String expression containing the name of the section where the key setting is found. GetAllSettings returns a Variant whose contents is a two-dimensional array of strings containing all the key settings in the specified section and their corresponding values.

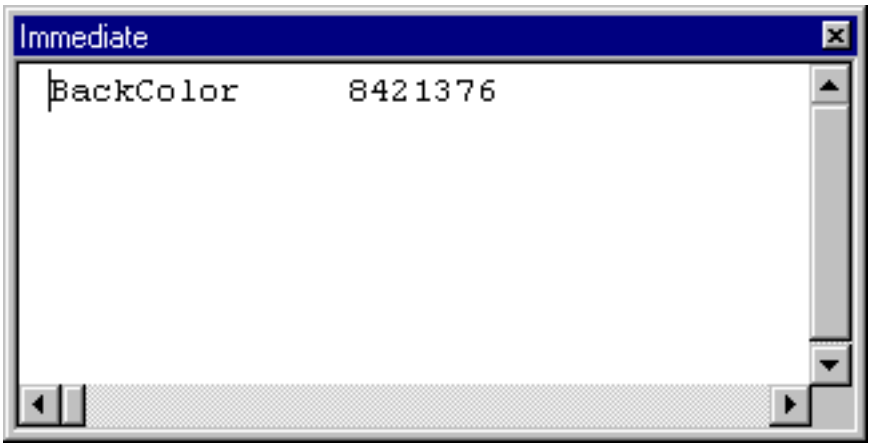
The tricky part of this Function is that the return value is a two dimensional array consisting of a key name and its value. Working with a two dimensional array isn't difficult however, and this code can be used to 'read' the array, and display the contents in the Immediate Window. You'll find almost this identical code in Visual Basic Help.

```
Dim MySettings As Variant, intSettings As Integer

MySettings = GetAllSettings("SmileyApplication", "Startup")

For intSettings = LBound(MySettings, 1) To UBound(MySettings, 1)
  Debug.Print MySettings(intSettings, 0), MySettings(intSettings, 1)
Next intSettings
```

Once **GetAllSettings** executes and returns a two dimensional array, we can use the Lbound and Ubound functions, in conjunction with a For…Next Loop to move through the elements of the Array and display the contents in the Immediate Window.



Summary

Writing to and reading information from the Windows Registry gives your program the ability to save important pieces of data. Most importantly, you can 'hide' this data from the user. I hope that I've taken some of the mystery out of the process, and that you'll feel free to use the Registry as often as you deem fit.