January, 2002

Working with the Registry

The Registry class makes it easy

By Tamar E. Granor

The Windows Registry is full of information about the user, his or her settings, the installed software and the machine itself. It's often useful to be able to extract some of that information in an application or to store some information there. For example, in my article "Make Them Hear You" in the November issue of FoxPro Advisor, I stored and extracted information about system sounds and application-specific sounds.

There's a set of API functions that work with the Registry, but using them can be pretty complex. Fortunately, one of the class libraries that comes with VFP 6 and 7 is Registry.VCX, which contains the class Registry. It's in the FFC subdirectory of the VFP installation. (VFP 5 also contains a Registry class. It's PRG-based, however. It's in the Samples\Classes subdirectory.)

The Registry class makes it much easier to work with the Registry. It has methods for all of the operations you're likely to want to perform. The methods make the appropriate API calls and handle errors that occur.

The Registry Structure

The Registry has a hierarchical structure branching out from a number of root nodes. Below each root, it's organized into keys and values. You can think of it as a database (in fact, it is a database, though not a relational one like VFP), where each key is a field name. However, unlike a relational database, a field may have multiple named values. Figure 1 shows the part of the Registry (using Windows 2000's RegEdt32.EXE Registry Editor) that contains settings from VFP 7's Options dialog.

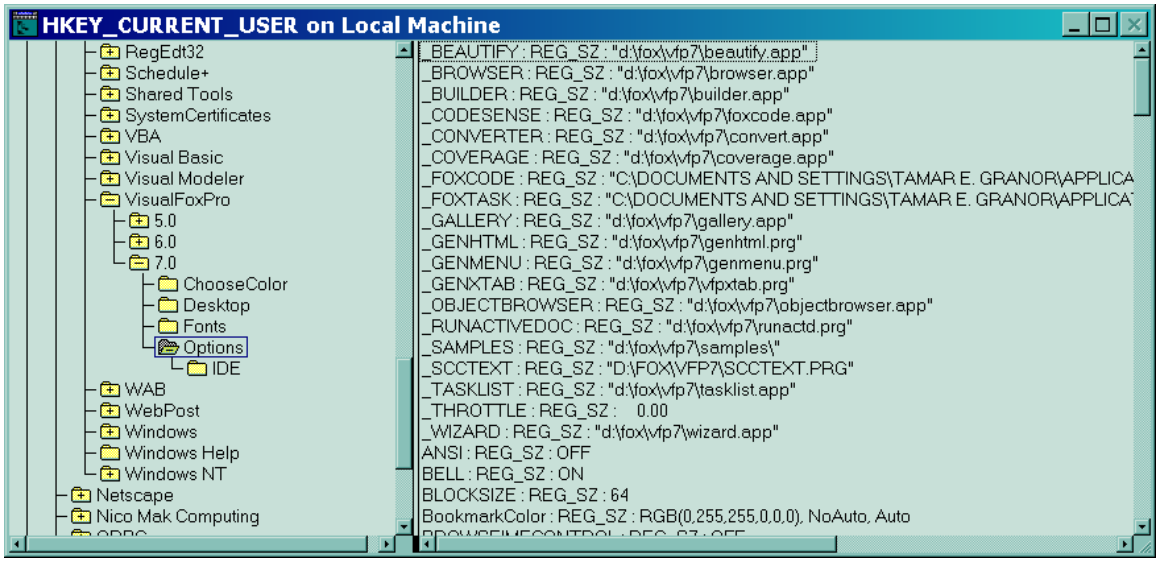

Figure 1. Exploring the Registry—This portion of the HKEY_CURRENT_USER group contains settings from VFP 7's Options dialog.

The full path to the highlighted item is HKEY_CURRENT_USER\Software\Microsoft\VisualFoxPro\6.0\Options. It contains quite a few values, shown in the right pane.

The names of the root nodes (like HKEY_CURRENT_USER) are actually constants representing large, negative numbers. You'll find their values in Registry.H in the same directory as the Registry class. Otherwise, the names you see in the registry editor are the names you use to access the various items.

The Registry Class

To use the registry class, you need to instantiate it, like this:

```
oRegistry = NewObject( "Registry", ;
                       HOME() + "FFC\Registry.VCX")
```

The class has a number of methods meant for public use. There are also a number of other methods designed for internal use by the "public" methods, but for some reason, they're not marked as protected. The methods intended for public use are reasonably well documented in the Help entry "Registry Access Foundation Class." We'll look at a few of them here to show how you can accomplish some basic tasks and to deal with some of the not-so-obvious things.

Does a Key Exist?

The first task is determining whether a particular Registry key exists. The IsKey method is designed specifically for this purpose. It takes two parameters: the key we're looking for and the root node. For example, to check for the key mentioned above, we'd called the method like this:

```
#DEFINE HKEY_CURRENT_USER -2147483647
lExists = oRegistry.IsKey( ;
    "Software\Microsoft\VisualFoxPro\7.0\Options", ;
    HKEY_CURRENT_USER)
```

IsKey returns .T. if it finds the key and .F. otherwise.

Getting Key Values

Now, let's retrieve the value of a key. When you have a multi-valued key like the VFP 7 Options key, you can ask for specific items. The GetRegKey method retrieves the value. To use it, you must first create a variable to hold the result and then pass that variable by reference. (Note that the Registry class can only deal with character values. Only those items that show a type of REG_SZ in the Registry Editor can be accessed through this class. Also, be aware that even numeric values are returned as characters.) Like many of the Registry class's methods, GetRegKey returns an error code. A return value of 0 indicates success; a non-zero value indicates failure.

To get the name of the current builder application (_BUILDER in VFP), use code like this:

```
#DEFINE HKEY_CURRENT_USER -2147483647
cValue = ""
nError = oRegistry.GetRegKey("_Builder", @cVal, ;
   "Software\Microsoft\VisualFoxPro\7.0\Options", ;
   HKEY_CURRENT_USER)
```

The first parameter indicates the name of the item you want to retrieve. In Figure 1, it's the string that precedes the colon in the right pane. The second parameter is the variable to hold the result. The third parameter is the key name and the final parameter is the root node (HKEY_CURRENT_USER, in this case).

The Registry class provides an alternate way to retrieve the same information. This approach requires two steps. First, you "open" the key, making it the current focus of attention. (Opening a key is analogous to selecting a work area in VFP.) Then, you can retrieve the

value you want without having to specify the name of the key. The following code gives the same results as the previous example:

```
#DEFINE HKEY_CURRENT_USER –2147483647
cValue = ""
nError = oRegistry.OpenKey( ;
   "Software\Microsoft\VisualFoxPro\7.0\Options", ;
   HKEY_CURRENT_USER, .f.)
IF nError = 0
   nError = oRegistry.GetKeyValue( "_Builder", @cVAl)
ENDIF
```

What's the benefit of having two ways to do this? Not a whole lot, though I suspect that when you're retrieving a whole series of values from the same key, the second approach is faster, since you don't have to go hunting for the key each time. However, there are some methods that require you to open the key first.

Suppose you want to get all the values for a particular key. The EnumKeyValues method lets you put the list of names and values into an array. EnumKeyValues has a single parameter, an array. If you pass a one-dimensional array, it retrieves only the names for the values. If you pass a two-dimensional array, however, EnumKeyValues gets both the names and values. Whichever way you do it, though, the key you're interested in must be opened first.

For example, this code gets the full list of names and values for the Options key:

```
#DEFINE HKEY_CURRENT_USER –2147483647
DIMENSION aOptionsValues[1,2]
nError = oRegistry.OpenKey( ;
   "Software\Microsoft\VisualFoxPro\7.0\Options", ;
   HKEY_CURRENT_USER, .f.)
IF nError = 0
   nError = oRegistry.EnumKeyValues( @aOptionsValues )
ENDIF
```

Here's the first few rows of aOptionsValues after running this code:

```
   (    1,    1)            C           "LastProject"
   (    1,    2)            C           "1"
   (    2,    1)            C           "TALK"
   (    2,    2)            C           "ON"
   (    3,    1)            C           "StatusBar"
   (    3,    2)            C           "1"
```

Sometimes what you want isn't a list of values, it's a list of subkeys. The EnumKeys method fills an array with the subkeys for the open key. Like EnumKeyValues, it requires the key of interest to be open

first. No matter what kind of array you pass, EnumKeys creates a one-dimensional array with one item for each subkey.

This code gets the list of subkeys for the Options key:

```
#DEFINE HKEY_CURRENT_USER -2147483647
DIMENSION aOptionsSubkeys[1]
nError = oRegistry.OpenKey( ;
   "Software\Microsoft\VisualFoxPro\7.0\Options", ;
   HKEY_CURRENT_USER, .f.)
IF nError = 0
   nError = oRegistry.EnumKeys( @aOptionsSubkeys )
ENDIF
```

Here's the total contents of aOptionsSubKeys after this code:

```
         (   1)                     C           "IDE"
         (   2)                     C           "IntelliDrop"
```

To get a full list of all the values for a key and its subkeys, you can use EnumKeys and EnumKeyValues in a recursive routine. The program in Listing 1 (on this month's Professional Resource CD as aGetKeys.PRG) fills an array with this information.

Listing 1. Retrieving all subkeys and values – This recursive program starts with a key and fills an array with all the values for that key and its subkeys.

```
* aGetKeys.PRG
* Copyright 2001, Tamar E. Granor
* Get all keys and subkeys of a specified key
* with their values.
* This routine is recursive.
LPARAMETERS aAllKeys, cKey, nRoot, oRegistry
  * aAllKeys = the array to hold the keys and values
  * cKey = the key to start with
  * nRoot = the root node in the Registry
  * oRegistry = object reference to an instance of
  *             the Registry class

* Check parameters
ASSERT TYPE( "aAllKeys[1]") <> "U" ;
  MESSAGE "aGetKeys: First parameter must be an array"
IF TYPE( "aAllKeys[1]") = "U"
  ERROR 232,"aAllKeys"
  RETURN -1
ENDIF

ASSERT VARTYPE( cKey ) = "C" AND NOT EMPTY( cKey );
  MESSAGE "aGetKeys: Second parameter (cKey) must be " + ;
          "character and not empty"
IF VARTYPE( cKey ) <> "C" OR EMPTY( cKey )
  ERROR 11
  RETURN -1
```

```
ENDIF

ASSERT VARTYPE( nRoot ) = "N" AND NOT EMPTY( nRoot) ;
  MESSAGE "aGetKeys: Third parameter (nRoot) must be " + ;
          "numeric and not empty"
IF VARTYPE( nRoot ) <> "N" OR EMPTY( nRoot)
  ERROR 11
  RETURN -1
ENDIF

* Check for a registry object
IF VARTYPE( oRegistry ) <> "O"
  oRegistry = NEWOBJECT( "Registry", ;
                         HOME() + "FFC\Registry")
ENDIF

LOCAL aValues, aKeys
LOCAL nCountSoFar, nValues, nItem, nKey

* Open the key and get started
WITH oRegistry
  IF .OpenKey( cKey, nRoot ) = 0
    * Get the values at this level and
    * copy them to the master array
    IF .EnumKeyValues( @aValues ) = 0
      nCountSoFar = ALEN( aAllKeys, 1)
      IF EMPTY(aAllKeys[1,1])
        nCountSoFar = 0
      ENDIF
      nValues = ALEN( aValues, 1)
      DIMENSION aAllKeys[ nCountSoFar + nValues, 3 ]
      FOR nItem = 1 TO nValues
        aAllKeys[ nCountSoFar + nItem, 1] = cKey
        aAllKeys[ nCountSoFar + nItem, 2] = ;
           aValues[ nItem, 1]
        aAllKeys[ nCountSoFar + nItem, 3] = ;
           aValues[ nItem, 2]
      ENDFOR
    ENDIF

    * Now get the keys at this level
    IF .EnumKeys( @aKeys ) = 0
      nKeyCount = ALEN( aKeys )
      FOR nKey = 1 TO nKeyCount
        aGetKeys( @aAllKeys, ;
                  ADDBS(cKey) + aKeys[ nKey ], ;
                  nRoot, oRegistry )
      ENDFOR
    ENDIF
  ENDIF
ENDWITH

RETURN ALEN(aAllKeys, 1)
```

Changing Registry Data

Now that we know how to retrieve all the information in the Registry, the next step is to be able to change the values there and to add new ones. The SetRegKey method lets you set a value for a key. This line creates a new Registry key (HKEY_CURRENT_USER\Software\Test) and creates an item called First with a value of "abc".

```
nError = oRegistry.SetRegKey( "First", "abc", "Software\Test", ;
            -2147483647, .T.)
```

The first four parameters are the same as for GetRegKey, except that the second parameter is the actual value to store (though you can, of course, use a variable to pass it). The last parameter indicates whether the key should be created if it doesn't already exist.

As with retrieving a value, you can also set a value in two steps, using OpenKey first, then using SetKeyValue. Here's the same example, using the two-step process.

```
#DEFINE HKEY_CURRENT_USER -2147483647
nError = oRegistry.OpenKey( ;
   "Software\Microsoft\VisualFoxPro\7.0\Options", ;
   HKEY_CURRENT_USER, .t.)
IF nError = 0
   nError = oRegistry.SetKeyValue( "First", "abc")
ENDIF
```

Note the third parameter to OpenKey. As in GetRegKey, this parameter determines whether the key should be created if it doesn't exist.

Handling Nameless Values

As you explore the Registry, you'll probably notice that some keys have a value with no name. For example, the key HKEY_CURRENT_USER\AppEvents\EventLabels\.Default has a single value ("Default Beep", on my machine), shown in Figure 2. In the Registry Editor, the value is preceded by the string "<No Name>".

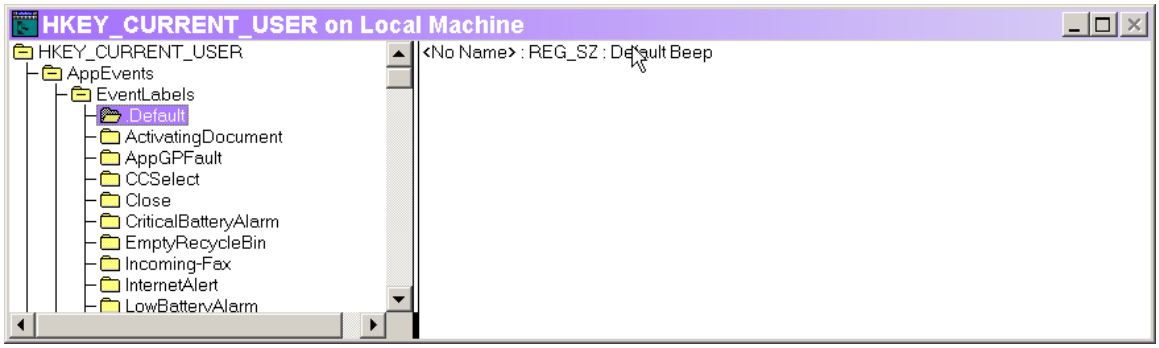

Figure 2. No name - Some keys have an unnamed value. To set this value, you need to use a string set to .null. for the name parameter of SetRegKey.

Since GetRegKey and SetRegKey require the name of the value to retrieve, a nameless value presents a problem. The solution is slightly different for retrieving the value than for storing it. With GetRegKey, the secret is to pass the empty string for the name, like this:

```
nError = oRegistry.GetRegKey( '', @cSound, ;
   "AppEvents\EventLabels\.Default", -2147483647)
```

SetRegKey uses a different approach. In that case, you need to pass the .null. value. However, it's not sufficient to pass .null. as a constant – you need to create a character variable and set it to .null., like this:

```
#DEFINE HKEY_CURRENT_USER -2147483647
cNull = ''
cNull = .null.
nError = oRegistry.SetRegKey( cNull, cSound, ;
   "AppEvents\EventLabels\.Default", HKEY_CURRENT_USER)
```

Removing Registry Keys

Finally, you may want to remove a key from the Registry. The DeleteKey method does that. It requires two parameters – the numeric root node and the full key except for the root node. For example, to delete the Software\Test key created earlier, use this code:

```
nError = oRegistry.DeleteKey( -2147483647, ;
   "Software\Test")
```

There's one complication when deleting Registry keys. You can't delete a key that has any subkeys, so you have to use a recursive routine that drills down into the key, deleting all the subkeys before deleting the current key. This month's Professional Resource CD contains DrillDownDelete.PRG, a routine that deletes a key and its subkeys.

```
* PROCEDURE DrillDownDelete
* Copyright 2001, Tamar E. Granor
```

```
* Drill down into a key, deleting all subkeys,
* then deleting the key itself.
* This procedure is recursive.

LPARAMETERS oRegistry, cKey, nRegKey
  * oRegistry = object reference to Registry object
  * cKey = key to be deleted
  * nRegKey = hive from which key is to be deleted

* Check parameters
ASSERT VARTYPE(oRegistry) = "O" ;
  MESSAGE "DrillDownDelete: First parameter is " + ;
          "required pointer to registry object"

IF VARTYPE(oRegistry) <> "O"
  ERROR 11
  RETURN .f.
ENDIF

ASSERT VARTYPE(cKey) = "C" AND NOT EMPTY(cKey) ;
  MESSAGE "DrillDownDelete: Second parameter is " + ;
          "required registry key"

IF VARTYPE(cKey) <> "C" or EMPTY(cKey)
  ERROR 11
  RETURN .f.
ENDIF

ASSERT VARTYPE(nRegKey) = "N" AND NOT EMPTY(nRegKey) ;
  MESSAGE "DrillDownDelete: Third parameter is " + ;
          "required registry node"

IF VARTYPE(nRegKey) <> "N" OR EMPTY(nRegKey)
  ERROR 11
  RETURN .F.
ENDIF

* set up an array to hold the list of subkeys
LOCAL aKeys[1], lSuccess, nSubkeyCount, nKey

lSuccess = .t.

WITH oRegistry
  IF .OpenKey(cKey, HKEY_CURRENT_USER) = 0
    IF .EnumKeys(@aKeys) = 0

      * Loop through subkeys
      nSubkeyCount = ALEN(aKeys, 1)
      FOR nKey = 1 TO nSubKeyCount
        lSuccess = DrillDownDelete( oRegistry, ;
                       cKey + "\" + aKeys[nKey], ;
                       nRegKey )
      ENDFOR
    ENDIF
    IF .DeleteKey(nRegKey, cKey) = 0
```

```
      lSuccess = lSuccess and .T.
    ELSE
      lSuccess = .F.
    ENDIF
  ELSE
    lSuccess = .F.
  ENDIF
ENDWITH

RETURN lSuccess
```

Go Forth and Register

As always when working with the Registry, be careful. Changing the wrong thing in the Registry can really foul up your system. On the other hand, experiment with reading values all you want. You can't cause any damage that way.

You may also want to subclass the Registry class to make it even easier to use. The Registry.VCX class library contains several subclasses that have methods for exploring particular parts of the Registry. Contributing Editor Steve Sawyer suggests subclasses for particular root nodes, as well as hard-coding options that you always use. (For example, he sets SetRegKey to always create a key if it doesn't exist.) You might also subclass for the tasks you use the Registry for all the time, creating methods that do your common tasks, so you don't have to remember the parameters for the various methods. Your custom methods might include some to set and extract options for your applications.

Although working with the Registry can seem quite daunting at first, the Registry class makes it pretty straightforward. A little experimentation with its methods should give you a pretty good feel for which ones to use when.