PINNACLE
FoxTalk
Solutions for Microsoft® FoxPro® and Visual FoxPro® Developers

This is an exclusive supplement for *FoxTalk* subscribers. For more information about *FoxTalk*, call us at **1-800-788-1900** or visit our Web site at **www.pinpub.com/foxtalk.**

Extended Article

Registering Users' Interests

Robin Dewson

It's the bane of all users' lives—they spend the first part of their day positioning the forms on the screen where they find them ideal, and then the system crashes. Then they have to go through the same process, setting up their system just the way they want it. Wouldn't it be nice if, when they start up their application, the screen displayed just as they had it arranged when they left? How useful would it be to allow each user to have their own individual screen set up, so that they decided what was to be displayed at startup?

GOOD use of the Registry can enable applications to be user-specific, even if more than one user uses the same machine. I recently came across a good example of this in a Fox 2.6 application I'm converting to VFP. The system is in use most of the day, and it's constantly monitoring purchase orders with deliveries and using this information to analyze how efficient a supplier is. Deliveries come in to companies 24 hours a day, and so there's shift work, which means there's a great deal of desk re-utilization.

Through use of saving, retrieving, and manipulating values in the Registry, and good placement of these values within the Registry in HKEY_CURRENT_USER, you can make your application customize itself for each user. The ability to do this requires a small amount of knowledge on how the Windows Registry works and which part of the Registry fits your needs.

The Windows Registry is composed of five sections, and each is used for a different function. The HKEY_CURRENT_USER section will hold the necessary entries required to create unique user individuality, and it's the section I'll be concentrating on. I personally use REGEDT32.EXE, which can be found in the WINNT\System32 directory, but the older REGEDIT.EXE also has its uses if you want to scan all five Registry sections at once (see Figure 1).

Save me, please

First of all, it was necessary to have the ability to save necessary information to the Registry, as this would make testing of the other features possible without having to manually add keys and values. In the accompanying file available in the Subscriber Downloads at www.pinpub.com/foxtalk, I've included a sample app and a single form that I used when building and testing this class, which is useful for just looking at what needs to be set up for each function. What goes where is very important, and I've included a button that points things out along the way. It's crude but effective, and useful as a basic Registry editor (see Figure 2).

When a form is closed—either when the application shuts down or when the user closes the form to move on—the height, left, top, and width values of the form are saved to the Registry. Also, don't forget to save the name

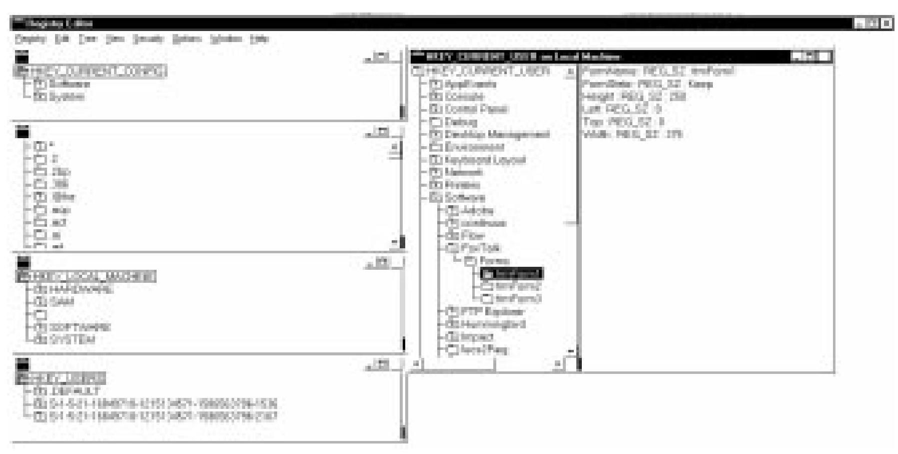

Figure 1. What the Registry would look like after running the sample application and exiting with all three forms open.

property, as this will be used in macro substitution later on, if the form is to be automatically opened. At this point, the Registry should be updated to state that the form is Closed. More manipulation of the form state value will occur later—specifically, when the application closes—but for now, let's just keep it marked as "Closed". I placed this code in my form's baseclass Unload event, so that the Registry would only be updated when the form was closing down. The SaveRegistry method from the Registry class is called for each property—six properties for each form.

All methods in the Registry class attempt to load the DLLs that will be required for all functions that could be used, but of course, this is only completed once per instantiation. A Registry key must now be open before a save can take place. The following code demonstrates how simple it is to open a key . . . if you get it right.

```
* Opens a Registry key
LOCAL nSubKey,nErrorNumber
nSubKey = 0

* Ensure that we have already run method
* SetUserKeyNumber. If not, then set the
* UserKeyNumber to HKEY_CURRENT_USER.
IF TYPE("This.UserKeyNumber") <> "N" ;
  OR EMPTY(This.UserKeyNumber)
  This.UserKeyNumber = -2147483647
ENDIF

* If a key has to be created, then do so.
IF THIS.lCreateKey
  * Create a Registry key.
  * Multiple keys won't be created.
  nErrorNumber = RegCreateKey(This.UserKeyNumber, ;
    This.KeyPath,@nSubKey)
ELSE
  * Try to open Registry key.
  nErrorNumber = RegOpenKey(This.UserKeyNumber, ;
    This.KeyPath,@nSubKey)
ENDIF

IF nErrorNumber <> SUCCESS
  RETURN nErrorNumber
ENDIF

* A unique key number will be returned, so we
* need to store this for later use.
THIS.nCurrentKey = nSubKey

RETURN SUCCESS
```

As you can see, RegCreateKey and RegOpenKey use the same parameters. The first parameter is the numeric equivalent of the Registry hive, in this instance HKEY_CURRENT_USER. The KeyPath is the point at which you wish to open the key. KeyValues are then placed below this level. To clarify the issue, think of the Registry as a form: The KeyPath is the actual form container and the KeyValues are the properties. Once the key is open, the application is now able to save the property and its value to the Registry.

```
LOCAL nValueSize,nErrorNumber,sOptionValue

DO CASE
CASE TYPE("THIS.nCurrentKey")<>'N';
  OR THIS.nCurrentKey = 0
```

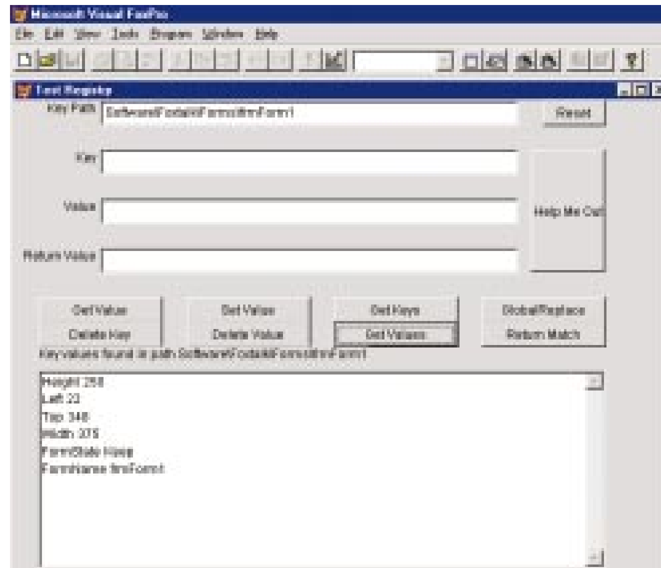

Figure 2. The form used for testing out the Registry class, but it can also be used as a crude Registry editor.

```
  RETURN 1
CASE TYPE("This.OptionName") <>"C"
  RETURN 2
CASE EMPTY(This.OptionName) ;
  OR EMPTY(This.OptionValue)
  RETURN 2
ENDCASE

IF TYPE("This.OptionValue") = "C"
  sOptionValue = This.OptionValue+CHR(0)
ELSE
  sOptionValue = ALLTRIM(STR(This.OptionValue));
  +CHR(0)
ENDIF

nValueSize = LEN(sOptionValue)
nErrorNumber = RegSetValueEx(THIS.nCurrentKey, ;
  This.OptionName,0, REG_SZ,sOptionValue, ;
  nValueSize)

* Check for error
IF nErrorNumber <> SUCCESS
    RETURN nErrorNumber
ENDIF

RETURN SUCCESS
```

Hurdling the fences

The first part of the code is the validation that the right parameters have been set and passed to the class. Once everything is well, you'll see that CHR(0) is added to the end of the value to be written to the Registry. This is because the Registry expects values to be null-terminated. And this is where the first problem comes up when writing to the Registry using Visual FoxPro.

If a property is numeric, as most are with the properties that need to be saved, adding a CHR(0) to the end gives an Operator/Operand type mismatch error. This means that to use VFP and the Registry, all values have to be stored as strings. If you desperately wish the Registry class to return the correct type, then you'd have to prefix the string by a unique set of characters (perhaps

a tilde) and then the type of string. You could then manipulate the string to return the right value. I feel that this is a lot of extra work when the calling program to the Registry class knows that it will always have a string returned and therefore knows when to use a VAL.

Once the value is null-terminated, the length of the string is calculated and passed as a parameter to RegSetValueEx. REG_SZ defines that a string is being passed to the DLL call, and so it should deal with the parameter in as a string.

The OptionName parameter contains the property name to save—in this case, it would have Height, Top, Left, Width, FormName, and FormState all passed individually so that each can save the necessary value.

Get back

Now that the class saves values to the Registry, it has to get them back. Again, the key is opened, which then allows the key to be retrieved.

```
* Obtains a value from a Registry key

LOCAL lpReserved, lpType, lpData, lpcbData, nErrorNumber
lpReserved = 0
lpType = 0
lpData = SPACE(256)
lpcbData = LEN(lpData)

DO CASE
CASE TYPE("THIS.nCurrentKey")<>'N' ;
  OR THIS.nCurrentKey = 0
  RETURN ERROR_BADKEY
CASE TYPE("This.OptionName") <>"C"
  RETURN ERROR_BADPARM
ENDCASE

nErrorNumber=RegQueryValueEx(THIS.nCurrentKey,;
  This.OptionName, lpReserved, @lpType,;
  @lpData, @lpcbData)

* Check for error
DO CASE
CASE nErrorNumber = SUCCESS
  * Make sure this is a string data type
  IF lpType <> REG_SZ
    RETURN 5
  ENDIF

  * Remove the Null at the end
  This.OptionValue = LEFT(lpData,lpcbData-1)
  RETURN SUCCESS
* Oh no! No key found. Is a key to be created
* or must an error code be returned?
CASE nErrorNumber = NO_KEY_FOUND
  IF This.lCreateValue
    RETURN This.SetKeyValue()
  ELSE
    RETURN NO_KEY_FOUND
  ENDIF

OTHERWISE
  RETURN m.nErrorNumber

ENDCASE
```

As you can see, there's a bit more to retrieving a value, compared to saving one. However, it's not as bad as it looks. Again, the Registry DLL call is a very fussy animal, and the Registry class has to cater to its fussiness. LpData will hold the returned string. So, to avoid errors, it has to be pre-loaded with spaces, so that the value can be placed in safely. Don't forget that the string will be null-terminated on a successful return from the DLL call, so this has to be removed from the end of the string. If no key is found, and the option of creating the key when no key is found is set, then, of course, this will be called. The bulk of the Registry class is now complete. However, this is only provides a tool that I need with respect to the bigger picture of what I'm trying to achieve.

Opening up for the day

Opening forms when a user first starts the system is theoretically relatively easy, but it does involve some work and traversing of the Registry. The code for retrieving the values should be placed just before READ EVENTS is issued. When a form is initially opened from the menu, the FormState of Open is saved in to the Registry. If a user closes the form, the FormState is altered to Closed. As you'll see a bit later, when the *application* is closed, the FormState is altered from OPEN and set to KEEP. The FormState needs to be altered from Open to Keep due to the two different ways a form is Unloaded. If a user closes the form voluntarily, for example, when they've finished with it, the form Unload event fires. In this method, there's code to alter the FormState from Open to Closed.

However, when the user decides to close the application *before* the form is unloaded, the Registry is traversed, and all forms that have a FormState of Open are altered to Keep. Then, when the form unloads when the application is in its final stages of closing and destroying itself and all objects, there will be a point when the form's Unload event fires. To keep those closed by the application and those closed by the user separate, the Unload won't find a state of Open and so the FormState will remain as Keep.

This task uses the Registry class's GlobalReplace function. The sole purpose of this function is to search a key path and look for a key's value. If it finds this value, then alter it to the value you wish. The GlobalReplace function, first of all, after opening up the Key, calls a hidden method called GetAllKeys. This method calls the RegEnumKeyEx DLL continuously, until an end-of-file situation occurs. It does this by starting at the root key of the opened KeyPath, knowing that the first key immediately below the root has an alias key number of 0. Each subsequent key will be incremental of the previous key number. So, by performing a loop and incrementing the key number, this method can keep going until a number isn't found. One thing to remember is that these numbers are relative and not static. Therefore, if you delete key number 4, key number 5 then becomes 4, and so forth, so there will never be any gaps.

Now that all the keys have been created and placed

into an array, a similar scenario can be completed to return all the key values. As each key value is returned, don't forget that the value is null-terminated, and a substring is required to remove that null. GlobalReplace then performs a simple loop comparing what's in each key value with the value to be tested, and when there's a match, a simple Save to the Registry is performed. In the sample application's case, this is where FormState of Open is altered to Keep.

Once this is complete, the application can now move on and open up the forms. You'll notice that I let the developer choose when to Reset the arrays in the Registry to give greater flexibility.

```
LOCAL nPos,nRet

* If the app crashed out from our application, there
* would be forms that were open at the time,
* POSSIBLY (depends on whether it was a clean crash
* or not) set to Open. Let's move them to Keep to
* keep things simple...

* The following procedure enumerates around the
* Registry and alters FormState of Open to Keep.
* This will then mean that the application is only
* looking for one Registry value.
This.ChangeOpenToKeep


* OReg is Registry class placed on the baseclass form
* Reset the arrays first of all.
This.OReg.Reset


* Look down Registry to the following KeyPath,
* and details
This.OReg.KeyPath = "Software\FoxTalk\forms"
This.OReg.lCreateKey = .F.
This.OReg.OptionName = 'FormState'
This.OReg.OptionValue = "Keep"
This.OReg.OptionValueToReturn = "FormName"

* Let's bring back all Key entries that match the
* above search criteria.
nRet = This.oReg.ReturnMatch()

* Providing we have a match, then let's display them...
* Check the array created and the Macro substitute.
IF !EMPTY(This.oReg.aMatchedEntries[1])
  FOR nPos = 1 to ALEN(This.oReg.aMatchedEntries,1)
    * Of course, if you create your forms as objects,
     * then alter the DO FORMS to CREATEOBJECT statements.
    DO FORM (This.oReg.aMatchedEntries(nPos))
  NEXT
ENDIF
```

Returning a match

The ReturnMatch function is similar to the GlobalReplace, but instead of replacing values, when a match is found, a value of a property is returned. OptionName will contain the Key you're looking at and comparing (in the application's case, this will be FormState); OptionValue is the value you're wishing to find—in this case, Keep. OptionValueToReturn will contain the name of the key in the Registry that the Registry class has to look for, and return the value of when a match is found—in this case, FormName. GetAllKeys and GetAllKeyValues methods are still used to return the necessary values for comparison. The array created in GetAllKeyValues is searched for a match on OptionName and OptionValue, and when found, the value of OptionValueToReturn is used to populate the array, aMatchedEntries. The class then returns to the application with this array populated with the form names that have to be automatically started. Loop around this array, and then just macro substitute. When saving the FormName to the Registry, an assumption is made that the Name property on the form is exactly the same as the filename saved on your drive. If this isn't the case and you save forms to your drive named differently than the expression in the Name property, then there has to be another property where the disk name is stored. This will then allow macro substitution of the returned matched value to be performed in the DO FORM statement.

Conclusion

This just shows one use of the Registry class. Used as I've just shown, this technique makes your application easier for your users to use. You can extend this further for them by saving record positioning information in the Registry for each form, so that if the form is auto loaded as just described, it can also go to the most recently used record. This would be impressive, especially after a crash (unless your network is as infallible as you are and it never has a crash!). You can even use this class to save details to the Registry if you're shipping a demo version of an application, and then modify the Registry when a purchase is made. Finally, every time a form is moved or resized, you could update the Registry so that if your application does crash, when it starts up, everything will be *exactly* where it was just before the crash.

The Registry is a powerful database, and when used wisely and correctly, it can be used to give an application an expert look and feel. ▲

Robin Dewson is a consultant for Stable Computers Ltd. He is currently converting a supplier analysis system for Rohbe Inc. from FoxPro 2.6 to Visual FoxPro 6.0. He has worked on a number of projects, including converting FoxPro systems to Visual Basic 5, and is experienced in Sybase, SQL Server 6.5, and Sybase SQL Anywhere. wonderbison@cix.compulink.co.uk, R_Dewson@Rohbe.com.