**Reading and Writing Registry Keys with Visual Basic**

Since its introduction in Windows 95, the registry has more often than not intimidated developers. On contemporary systems, it is a large, multi-megabyte structure whose organization is confusing and whose contents, more often than not, are either under-documented or actually undocumented. Typically, developers are advised not to "muck with" the registry, lest the most dire of consequences (application crashes, system failure, or even a system that is unable to boot) result.

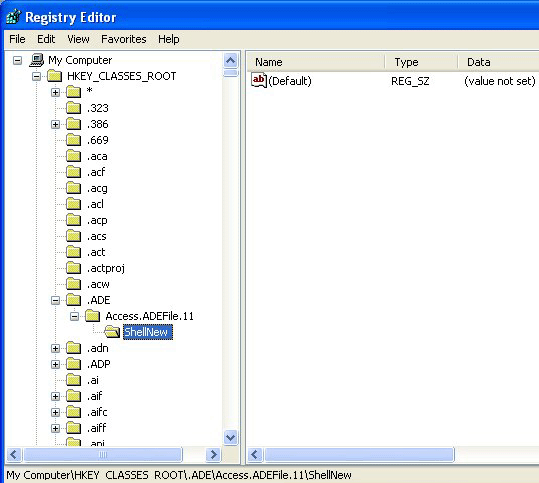
Yet although releases of Windows after Windows 95 have attempted to store critical information in other places (such as Active Directory and the IIS Metabase), the registry remains a critical, centralized database of system and application configuration information and settings. The ability to read from and write to the registry is an important component in the repertory of any developer.

Although the registry seems like an intimidating structure, in fact it is not. The registry is simply an inverse hierarchical tree-like structure, much like a file system. The major difference is that, while a file system consists of a single root per drive (the root directory or folder), the registry has either five or six roots, depending on the Windows version. These are:

* HKEY\_CLASSES\_ROOT, which stores information about file associations and COM objects.
* HKEY\_CURRENT\_USER, which stores the current user's configuration settings.
* HKEY\_LOCAL\_MACHINE, which stores the local system's configuration settings.
* HKEY\_USERS, which stores settings for the default and other users.
* HKEY\_CURRENT\_CONFIG, which stores the current system's hardware and software configuration.
* HKEY\_DYN\_DATA, a set of volatile (nonpermanent) keys used to store plug-and-play hardware configuration information and performance statistics on Windows 95/98/ME systems.
* HKEY\_PERFORMANCE\_DATA, a key used to support performance monitoring on Windows NT systems only.

There is also a difference in terminology. A file system consists of a root folder or root directory, which has one or more subfolders or subdirectories. Each subfolder or subdirectory in turn can have one or more subfolders or subdirectories, and so on. The registry, in contrast, consists of a number of root keys, each of which has multiple subkeys. Each subkey in turn can have one or more subkeys, and so on. In a file system, each folder can have files. In the registry, a key can have one or more values or value entries.

Figure 1 uses RegEdit, the registry editor, to illustrate the basic structure of the registry. One of the root keys,HKEY\_CLASSES\_ROOT, has numerous subkeys, one of which is .ADE. The HKEY\_CLASSES\_ROOT\.ADE key (notice the registry path, which describes the complete path to a registry key) in turn has a single subkey, Access.ADEFile.11. TheHKEY\_CLASSES\_ROOT\.ADE\Access.ADEFile.11 key in turn has a single subkey, ShellNew.HKEY\_CLASSES\_ROOT\.ADE\Access.ADEFile.11\ShellNew has a single value, a default (or unnamed) value whose value has not been set.

  
*Figure 1. The structure of the Windows registry.*

Although registry keys often serve as containers for holding one or more values, they can also be important in their own right. Figure 1 is a case in point. The ShellNew registry key is in part a flag; its presence indicates that a file of this type can be created using the New option on an Explorer context menu. In many cases such as these, it's the presence or absence of a key, rather than the values the key contains, that is significant.

This article will examine methods available in Visual Basic to perform basic operations upon registry keys, first using the basic language features found in Visual Basic itself, and then using the Win32 API.

**Visual Basic Language Support for the Registry**

Visual Basic itself includes four intrinsic functions (GetSetting, GetAllSettings, SaveSetting, and DeleteSetting) that support registry access. The documentation for these functions is somewhat confusing, since the functions themselves were designed to support both initialization (.ini) file access under 16-bit Windows and registry access under 32-bit Windows.

The functions themselves, however, suffer from a number of limitations that make them unsuitable for general-purpose access to the registry. One of the most important of these is that the functions allow you to read and write only to keys that are subkeys of HKEY\_CURRENT\_USER\Software\VB and VBA Program Settings. This has three major implications:

* The functions should be used only to store user-specific data. They should not be used to store application-specific or system-specific data, both of which should be stored in the subkeys of HKEY\_LOCAL\_MACHINE.
* The functions cannot be used to access keys throughout the registry, but can only be used to access a very small portion of registry keys and values.
* The registry is a dynamic structure. The exact contents of HKEY\_CURRENT\_USER, the per-branch of the registry, depends on the identity of the current user; there is a separate registry file for each user. This means that assuming that the relevant registry keys exist and attempting to access them just because the program has already been run once, for instance, can cause an application crash.

Given these limitations, as well as numerous others that we haven't discussed here, we recommend that you not use Visual Basic's intrinsic registry functions. Instead, registry access is best accomplished by using the Win32 API.

**Registry Access through the Win32 API**

By defining Win32 API functions using the Declare statement, you can access the entire registry and perform all operations that are supported by the Win32 API upon the registry. In order to do so, however, you need to define a number of constants in your code. First, each top-level registry key corresponds to an intrinsic constant, as follows:

Public Const HKEY\_CLASSES\_ROOT = &H80000000

Public Const HKEY\_CURRENT\_CONFIG = &H80000005

Public Const HKEY\_CURRENT\_USER = &H80000001

Public Const HKEY\_LOCAL\_MACHINE = &H80000002

Public Const HKEY\_USERS = &H80000003

Second, each of the registry functions returns a numeric code indicating its success or failure. A successful function call returns the following:

Public Const ERROR\_SUCCESS = 0&

Among the constants representing error codes are the following:

Public Const ERROR\_FILE\_NOT\_FOUND = 2& ' Registry path does not exist

Public Const ERROR\_ACCESS\_DENIED = 5& ' Requested permissions not available

Public Const ERROR\_INVALID\_HANDLE = 6& ' Invalid handle or top-level key

Public Const ERROR\_BAD\_NETPATH = 53 ' Network path not found

Public Const ERROR\_INVALID\_PARAMETER = 87 ' Bad parameter to a Win32 API function

Public Const ERROR\_CALL\_NOT\_IMPLEMENTED = 120& ' Function valid only in WinNT/2000?XP

Public Const ERROR\_INSUFFICIENT\_BUFFER = 122 ' Buffer too small to hold data

Public Const ERROR\_BAD\_PATHNAME = 161 ' Registry path does not exist

Public Const ERROR\_NO\_MORE\_ITEMS = 259& ' Invalid enumerated value

Public Const ERROR\_BADDB = 1009 ' Corrupted registry

Public Const ERROR\_BADKEY = 1010 ' Invalid registry key

Public Const ERROR\_CANTOPEN = 1011& ' Cannot open registry key

Public Const ERROR\_CANTREAD = 1012& ' Cannot read from registry key

Public Const ERROR\_CANTWRITE = 1013& ' Cannot write to registry key

Public Const ERROR\_REGISTRY\_RECOVERED = 1014& ' Recovery of part of registry successful

Public Const ERROR\_REGISTRY\_CORRUPT = 1015& ' Corrupted registry

Public Const ERROR\_REGISTRY\_IO\_FAILED = 1016& ' Input/output operation failed

Public Const ERROR\_NOT\_REGISTRY\_FILE = 1017& ' Input file not in registry file format

Public Const ERROR\_KEY\_DELETED = 1018& ' Key already deleted

Public Const ERROR\_KEY\_HAS\_CHILDREN = 1020& ' Key has subkeys & cannot be deleted

Third, some of the registry functions in the Win32 API have a parameter whose value is a security access mask indicating the rights that the current process must have in order for the function to execute successfully. The following are the possible values:

Public Const KEY\_CREATE\_LINK = &H20

Public Const KEY\_CREATE\_SUB\_KEY = &H4

Public Const KEY\_ENUMERATE\_SUB\_KEYS = &H8

Public Const KEY\_NOTIFY = &H10

Public Const KEY\_QUERY\_VALUE = &H1

Public Const KEY\_READ = ((STANDARD\_RIGHTS\_READ Or KEY\_QUERY\_VALUE

Or KEY\_ENUMERATE\_SUB\_KEYS Or KEY\_NOTIFY)

And (Not SYNCHRONIZE))

Public Const KEY\_SET\_VALUE = &H2

Public Const KEY\_WRITE = ((STANDARD\_RIGHTS\_WRITE Or KEY\_SET\_VALUE

Or KEY\_CREATE\_SUB\_KEY) And (Not SYNCHRONIZE))

Public Const KEY\_EXECUTE = ((KEY\_READ) And (Not SYNCHRONIZE))

Public Const KEY\_ALL\_ACCESS = ((STANDARD\_RIGHTS\_ALL Or KEY\_QUERY\_VALUE

Or KEY\_SET\_VALUE Or KEY\_CREATE\_SUB\_KEY

Or KEY\_ENUMERATE\_SUB\_KEYS Or KEY\_NOTIFY

Or KEY\_CREATE\_LINK) And (Not SYNCHRONIZE))

A bit of background about the operation of the registry functions in the Win32 is also important. Registry keys are system objects, and like any system object in the Win32 API, they are accessible through their handles. In other words, to work with a registry key, you must open it. In some form, the function that opens the key will return (usually in a parameter passed to the function by reference) a handle to the registry key, or hKey. With this background, and with these constants in place, you can proceed to access the registry API.

**Opening a Registry Key**

Opening the key is always the first step involved in working with a registry key (unless you intend to work with one of the top-level keys, which are always considered open). Once you open the key, you work with its handle, or hKey. We can take advantage of the fact that a successful open operation results in a valid handle to a registry key when determining whether a particular key exists, which is one of the more common operations performed on registry keys. For example, consider the following example function:

Private Function RegKeyExists(hKey As Long, sKeyPath As String) As Boolean

Dim lResult As Long

Dim hSubkey As Long

lResult = RegOpenKeyEx(hKey, sKeyPath, 0, KEY\_EXECUTE, hSubkey)

' Determine if handle is valid

If hSubkey <> 0 Then

RegKeyExists = True

RegCloseKey hKey

End If

End Function

This example function takes two parameters: a handle to a registry key (or a top-level registry key), and the path from the open key to the registry key to be opened. It then calls the RegOpenKeyEx function, which simply opens an existing key and returns its handle in phkResult, a parameter passed to the function by reference. Its syntax is:

Public Declare Function RegOpenKeyEx Lib "advapi32.dll" Alias "RegOpenKeyExA" \_

(ByVal hKey As Long, \_ ' Handle of already open key

ByVal lpSubKey As String, \_ ' Path from hKey to key to open

ByVal ulOptions As Long, \_ ' Reserved, must be 0

ByVal samDesired As Long, \_ ' SAM constant(s)

phkResult As Long) As Long ' Handle of newly opened key

The RegKeyExists function then examines the value of the RegOpenKeyEx function's phkResult argument to determine whether the key exists. If its value is non-zero, the phkResult argument has been assigned a valid handle value, indicating that the key exists. The example function can then return True.

Otherwise, it returns False. Instead of examining the value of the phkResult parameter, the example function could have examined the value returned by the RegOpenKeyEx function call. A value of ERROR\_SUCCESS indicates that the key exists, so the function should return True; any other value indicates that it does not exist.

The RegKeyExists function, if it determines that the key exists, calls the Win32 RegCloseKey function to close the opened key. Its syntax is:

RegCloseKey(hKey)

Where hKey is the handle to the registry key to close. Since handles to registry keys are system resources, calling theRegCloseKey function when the handle is no longer needed is good programming practice.

Rather than calling RegOpenKeyEx, an earlier and simpler version of the function, RegOpenKey, can be called instead.

The function was introduced as part of the Win16 API, when Windows was predominantly a single-user desktop operating system and system security was considered relatively unimportant, and so lacks the samDesired parameter. As a result, the function often returns ERROR\_ACCESS\_DENIED when the default security access mask used by the function exceeds the permissions required by the key. Because of this, using RegOpenKeyEx in the long run is simpler and more convenient.

**Opening or Creating a Registry Key**

Often when accessing the registry, you don't really care whether or not a key exists. Instead, you wish to open it if it exists and create it if it does not. This is often the case, for instance, when a new user first runs your application on a system configured to support multiple users. For this purpose, you use the RegCreateKeyEx function, which opens an existing registry key or, if the key does not exist, creates it. Its syntax is:

Public Declare Function RegCreateKeyEx Lib "advapi32.dll" Alias "RegCreateKeyExA" \_

(ByVal hKey As Long, \_ ' Handle of already open key

ByVal lpSubKey As String, \_ ' Path from hkey to key to open/create

ByVal Reserved As Long, \_ ' Reserved, must be 0

ByVal lpClass As String, \_ ' Reserved, must be a null string

ByVal dwOptions As Long, \_ ' Type of key, or backup/restore

ByVal samDesired As Long, \_ ' SAM constant(s)

lpSecurityAttributes As SECURITY\_ATTRIBUTES, \_

phkResult As Long, \_ ' Handle of opened/created key

lpdwDisposition As Long) As Long '

Although most of these parameters need no explanation, several are not so obvious. The dwOptions parameter determines the type of key to be created if lpSubKey does not exist. Possible values are:

Public Const REG\_OPTION\_BACKUP\_RESTORE = 4

Public Const REG\_OPTION\_NON\_VOLATILE = 0

Public Const REG\_OPTION\_VOLATILE = 1

Non-volatile keys are saved as part of the registry, while volatile keys are not. REG\_OPTION\_BACKUP\_RESTORE indicates thatRegCreateKeyEx is called as part of a registry backup/restore operation and if necessary overrides the samDesiredparameter to provide for the necessary access.

If the function succeeds, the operation it performed is indicated by the lphDisposition parameter, which is passed by reference to the function. Its value can be either of the following two constants:

Public Const REG\_CREATED\_NEW\_KEY = &H1 ' A new key was created

Public Const REG\_OPENED\_EXISTING\_KEY = &H2 ' An existing key was opened

The function also includes a parameter of type SECURITY\_ATTRIBUTES, which is defined as follows:

Public Type SECURITY\_ATTRIBUTES

nLength As Long

lpSecurityDescriptor As Long

bInheritHandle As Long

End Type

The SECURITY\_ATTRIBUTES type includes a security descriptor and determines whether the handle returned by the function call can be inherited. If we pass a Null as the value of the pointer to the security descriptor, a newly created registry key will be assigned the default security descriptor of the current process. Assuming that this is adequate, we can define this argument in either of two ways:

Dim sa As SECURITY\_ATTRIBUTES

or

Dim sa As SECURITY\_ATTRIBUTES

sa.nLength = Len(sa)

sa.bInheritHandle = CLng(True)

The code to either open or create a subkey of HKEY\_CURRENT\_USER named Software\MyCompany\MyCorporateApp is as follows:

Dim lResult As Long

Dim hTopKey As Long, hKey As Long, lDisposition As Long

Dim sRegPath As String

Dim sa As SECURITY\_ATTRIBUTES

hTopKey = HKEY\_CURRENT\_USER

sRegPath = "Software\MyCompany\MyCorporateApp"

sa.nLength = Len(sa)

sa.bInheritHandle = CLng(True)

lResult = RegCreateKeyEx(hTopKey, sRegPath, 0, vbNullString, REG\_OPTION\_NON\_VOLATILE, \_

KEY\_ALL\_ACCESS, sa, hKey, lDisposition)

If lResult = ERROR\_SUCCESS Then

If lDisposition = REG\_CREATED\_NEW\_KEY Then

' Assign default values

ElseIf lDisposition = REG\_OPENED\_EXISTING\_KEY Then

' Retrieve values from existing keys

End If

Else

MsgBox "Error " & lResult

End If

**Getting Information about Keys**

Once you've opened a key, you sometimes know what subkeys and values it contains. In other cases, you may not be completely sure about its keys and values. And sometimes, you have absolutely no idea what its subkeys and their values are. To help you handle situations in which you don't know the number and names of subkeys, you can call theRegQueryInfoKey function. Its syntax is:

Public Declare Function RegQueryInfoKey Lib "advapi32.dll" Alias "RegQueryInfoKeyA" \_

(ByVal hKey As Long, \_ ' Handle of key to query

ByVal lpClass As String, \_ ' Class (unsued except for remote keys)

lpcbClass As Long, \_ ' Length of class string (unused)

ByVal lpReserved As Long, \_ ' Reserved, must be 0

lpcSubKeys As Long, \_ ' Number of subkeys

lpcbMaxSubKeyLen As Long, \_ ' Length of longest subkey name

lpcbMaxClassLen As Long, \_ ' Length of longest class name (unused)

lpcValues As Long, \_ ' Number of values

lpcbMaxValueNameLen As Long, \_ ' Length of longest value name

lpcbMaxValueLen As Long, \_ ' Length of longest value

lpcbSecurityDescriptor As Long, \_ ' Security Descriptor

lpftLastWriteTime As FILETIME \_ ' Date/time of last write to key

) As Long

Note that most of these arguments are passed to the function by reference; that is, on return, their values are updated to reflect the results of the query. RegQueryInfoKey provides us with the following items of information:

* The number of subkeys. With this information, we can enumerate subkeys using the RegEnumKey function.
* The greatest number of characters in a subkey name. With this information, we can allocate a large enough buffer when calling RegEnumKey to retrieve the complete name of each subkey. Each of these subkeys can then be opened using the RegOpenKeyEx function.
* The number of values. With this information, we can enumerate values using the RegEnumValue function.
* The greatest number of characters in a named value. With this information, we can allocate a sufficiently large buffer when calling RegEnumValue to retrieve the complete name of each value.
* The greatest number of bytes in a value. With this information, we can allocate a large enough buffer when callingRegQueryValueEx to retrieve all the data belonging to each value.

Once you know the number of subkeys belonging to an open key, you can iterate them using the RegEnumKeyEx function. Its syntax is:

Public Declare Function RegEnumKeyEx Lib "advapi32.dll" Alias "RegEnumKeyExA" \_

(ByVal hKey As Long, \_ ' Handle of open key

ByVal dwIndex As Long, \_ ' Index of subkey

ByVal lpName As String, \_ ' Name of key

lpcbName As Long, \_ ' Length of name buffer

ByVal lpReserved As Long, \_ ' Reserved, must be 0

ByVal lpClass As String, \_ ' Class name (usually usused)

lpcbClass As Long, \_ ' Length of class name

lpftLastWriteTime As FILETIME \_ ' Date of last write

) As Long

The following code illustrates the use of RegQueryInfoKey, RegEnumKeyEx, and RegOpenKeyEx to open each of the subkeys of HKEY\_CURRENT\_USER\Control Panel:

Dim lResult As Long

Dim hKey As Long

Dim lNumKeys As Long ' Number of subkeys of open key

Dim lSubkeyNameLen As Long ' Length of the longest subkey name

Dim ft As FILETIME

Dim sRegPath As String

Dim sa As SECURITY\_ATTRIBUTES ' Use default security attributes of process

sa.nLength = Len(sa)

sa.bInheritHandle = CLng(True)

If RegOpenKeyEx(HKEY\_CURRENT\_USER, "Control Panel", 0, KEY\_READ, hKey) = ERROR\_SUCCESS Then

' Get subkey information

lResult = RegQueryInfoKey(hKey, vbNullString, 0, 0, lNumKeys, lSubkeyNameLen, \_

0, 0, 0, 0, 0, ft)

If lResult = ERROR\_SUCCESS Then

Dim sMsg As String

sMsg = "Key: HKEY\_CURRENT\_USER\Control Panel" & vbCrLf & vbCrLf

sMsg = sMsg & "Subkeys: " & lNumKeys & vbCrLf

sMsg = sMsg & "Longest subkey name: " & lSubkeyNameLen & vbCrLf

MsgBox sMsg

End If

' Use subkey information to enumerate subkeys

Dim lCtr As Long

Dim sBuffer As String

Dim lBuffer As Long

Dim hSubkey As Long

' Loop as many times as there are subkeys

For lCtr = 0 To lNumKeys - 1

' Initialize buffer to hold name to be 1 greater than maximum length

sBuffer = Space(lSubkeyNameLen) & Chr(0)

lBuffer = Len(sBuffer)

RegEnumKeyEx hKey, lCtr, sBuffer, lBuffer, 0, vbNullString, 0, ft

' Trim returned buffer to extract key name

sBuffer = Left(sBuffer, lBuffer)

sMsg = "Key " & lCtr & ": " & sBuffer & vbCrLf

' Open subkey

RegOpenKeyEx hKey, sBuffer, 0, KEY\_READ, hSubkey

sMsg = sMsg & " Returned handle " & Hex(hSubkey) & vbCrLf

MsgBox sMsg

RegCloseKey hSubkey

Next

RegCloseKey hKey

Else

MsgBox "Unable to open key: " & lResult

End If

Note that the code calls RegQueryInfoKey to determine the total number of subkeys and the number of characters in the longest subkey name. Since we're not interested in any of the other information that the function is able to retrieve, we simply provide default literal values (0 for Longs, the vbNullString constant for Strings) in place of the arguments we don't care about.

The loop that repeatedly calls RegEnumKeyEx has a starting index value of 0 and a maximum value of 1 less than the total number of subkeys. Failing to begin the loop at 0 will cause an error. For each call to RegEnumKeyEx, we initialize a buffer whose length is one greater than the maximum number of characters in a subkey name; this allows for the full subkey name to be written to the buffer, along with a terminating null character (or Chr(0)). When RegEnumKeyEx returns, lBufferis updated to reflect the actual number of characters in the subkey name without the terminating null character. The call to the intrinsic VB Left function discards any portion of the string that is unnecessary. Once we have extracted the name of the subkey, we can call RegOpenKeyEx to open that key.

**Deleting a Registry Key**

Deleting a registry key would appear to be a straightforward matter: you just call the RegDeleteKey function, whose syntax is:

Public Declare Function RegDeleteKey Lib "advapi32.dll" Alias "RegDeleteKeyA" \_

(ByVal hKey As Long, \_ ' Handle of open key

ByVal lpSubKey As String \_ ' Path from handle to key

) As Long

Here, hKey is a handle to an open key, and lpSubKey is a string containing the path to the subkey to be deleted. Since the second parameter cannot be a null string, the function requires that you delete a subkey of an open key. The complication, however, is that on Windows NT/2000/XP systems, the function fails if the subkey to be deleted itself contains any subkeys.

You can work around this problem in either of two ways:

* You can write code that uses recursion to iterate each of the child keys belonging to the key that you'd like to delete.
* You can call the Windows Shell library's SHDeleteKey function, which iterates and deletes the child keys before deleting the key itself.

Since relying on operating system services rather than writing code that duplicates system services is sound practice, calling SHDeleteKey is far preferable to writing your own code. The syntax of SHDeleteKey is:

Public Declare Function SHDeleteKey Lib "shlwapi.dll" Alias "SHDeleteKeyA" \_

(ByVal hKey As Long, ByVal pszSubKey As String) As Long

You could then delete the HKEY\_CURRENT\_USER\Software\MyCompany\MyCorporateApp key and its child keys with the following code:

Dim lResult As Long

Dim hKey As Long

If RegOpenKeyEx(HKEY\_CURRENT\_USER, "Software", 0, KEY\_ALL\_ACCESS, hKey) = ERROR\_SUCCESS Then

lResult = SHDeleteKey(hKey, "MyCompany")

RegCloseKey hKey

End If

[*Ron Petrusha*](http://www.windowsdevcenter.com/pub/au/255)*is the author and coauthor of many books, including "VBScript in a Nutshell."*