How to access environment variables

Contents

Why this article?

Sometimes we need to access some system configuration information that is stored in environment variables – such as the search path. These are stored in environment variables.

On other occasions we might want to provide a spawned application with some global information using environment variables. This can be done in two main ways:

Further Reading

For background material about environment variables, including a list of those created when Windows starts, see *Wilson Mar*'s *Environment Variables* article.

1. By setting some new environment variables and then spawning the new process which reads the information. The new process gets a copy of the parent's environment by default.
2. By creating an environment block programatically and passing that to the new process. The new process will get only the environment variables contained in the new environment block.

A little background information

Before we jump into writing some code to access and update environment variables, let us take a brief look at how Windows handles them.

When Windows starts a program it provides the program with a *copy* of its environment variables. The API functions that work with environment variables operate on this copy, *not on the original* environment variables. This means that any changes you make to environment variables only apply to the program's copy and *do not update* the underlying Window's environment variables – such changes are lost when your program terminates.

Any child processes started by your application gets, by default, a copy of *your program's* current environment, not that of Windows. This means that any changes you make to the environment before starting the child process *are* reflected in the child's own environment block. This default behaviour can be overridden by defining a custom block of environment variables to be passed to the child process. We discuss how to do this *below*.

One final point to note is that the block of memory allocated for environment variables is a fixed size. This means that you can't keep adding new environment variables to the block without limit. At some point the attempt to add a variable, or to store more data in an existing one, will fail. The size of the block created for a process depends on the number of environment variables and the size of the data to be passed to it. Back in the days of Windows 98 the block size was small (about 16Kb) and was incremented in 4Kb chunks. On modern Windows systems this size is massively increased.

How it's done

Now we have an understanding of how Windows handles environment variables, let us move on to look at how we work with them. Windows provides four API functions for accessing and updating environment variables:

1. **GetEnvironmentVariable**
   Returns the value of a given environment variable.
2. **SetEnvironmentVariable**
   Sets an environment variable's value, creating a new variable if necessary. This routine can also be used to delete an environment variable.
3. **GetEnvironmentStrings**
   Gets a list of all the environment variables available to a process.
4. **ExpandEnvironmentStrings**
   Replaces environment variables delimited by "%" characters in a string with the variable's value.

We will develop six Delphi routines that wrap these API calls plus one that can be used to create a new environment block for passing to a child process. They are:

1. ***GetEnvVarValue***
   Returns the value of a given environment variable.
2. ***SetEnvVarValue***
   Sets the value of the given environment variable.
3. ***DeleteEnvVar***
   Deletes the given environment variable.
4. ***GetAllEnvVars***
   Fills a string list with the names and values of all the program's environment variables.
5. ***ExpandEnvVars***
   Replaces all "%" delimited environment variables in a string with their values.
6. ***CreateEnvBlock***
   Creates a new environment block suitable for passing to a child process.

And so to the code:

GetEnvVarValue

This routine returns the value of a given environment variable (or returns the empty string if there is no variable with that name). Here's the definition:

```
function GetEnvVarValue(const VarName: string): string;
var
  BufSize: Integer;  // buffer size required for value
begin
  // Get required buffer size (inc. terminal #0)
  BufSize := GetEnvironmentVariable(
    PChar(VarName), nil, 0);
  if BufSize > 0 then
  begin
    // Read env var value into result string
    SetLength(Result, BufSize - 1);
    GetEnvironmentVariable(PChar(VarName),
      PChar(Result), BufSize);
  end
  else
    // No such environment variable
    Result := '';
end;
```

Listing 1

You'll notice that *GetEnvironmentVariable* is called twice: once to get the size of the required buffer and the second time to actually read the variable. This is a common Windows idiom.

SetEnvVarValue

This routine sets the value of an environment variable. If the variable doesn't already exist it is created. Zero is returned if all goes well, otherwise a Windows error code is returned. An error may occur if there is no room in the environment block for the new value. The implementation is very simple:

```
function SetEnvVarValue(const VarName,
  VarValue: string): Integer;
begin
  // Simply call API function
  if SetEnvironmentVariable(PChar(VarName),
    PChar(VarValue)) then
    Result := 0
  else
    Result := GetLastError;
end;
```

Listing 2

Remember, setting an environment variable only operates on the program's copy of the variable – it **does not** update the value of Window's copy of the variable. Changes are lost when the program terminates.

DeleteEnvVar

This routine deletes the given environment variable. Note that *SetEnvVarValue('')* has the same effect. Again, zero is returned on success and a Windows error code on error. The implementation is again simple:

```
function DeleteEnvVar(const VarName: string): Integer;
begin
  if SetEnvironmentVariable(PChar(VarName), nil) then
    Result := 0
  else
    Result := GetLastError;
end;
```

Listing 3

Once again, this function has **no effect** on Window's own copy of the environment variable.

GetAllEnvVars

This routine returns all of a program's environment variables in a string list. Each entry in the list is of the form `Name=Value`. You can use the *TStrings Names[]* and *Values[]* properties to extract the variable names and value from the string. The function returns the amount of space taken by the strings in the environment block. If you just want to know the size of the environment variables, pass a nil parameter. Here's the definition:

```
function GetAllEnvVars(const Vars: TStrings): Integer;
var
  PEnvVars: PChar;    // pointer to start of environment block
  PEnvEntry: PChar;   // pointer to an env string in block
begin
  // Clear the list
  if Assigned(Vars) then
    Vars.Clear;
  // Get reference to environment block for this process
  PEnvVars := GetEnvironmentStrings;
  if PEnvVars <> nil then
  begin
    // We have a block: extract strings from it
    // Env strings are #0 separated and list ends with #0#0
    PEnvEntry := PEnvVars;
    try
      while PEnvEntry^ <> #0 do
      begin
        if Assigned(Vars) then
          Vars.Add(PEnvEntry);
        Inc(PEnvEntry, StrLen(PEnvEntry) + 1);
      end;
      // Calculate length of block
      Result := (PEnvEntry - PEnvVars) + 1;
    finally
      // Dispose of the memory block
      Windows.FreeEnvironmentStrings(PEnvVars);
    end;
  end
  else
    // No block => zero length
    Result := 0;
end;
```

Listing 4

Often there will be an entry in the environment block not in the form `Name=Value` (it may have format `=Value`). The data can be read from the string list via its *Items[]* property, but not from the *Names[]* property.

ExpandEnvVars

This function takes as a parameter a string of text containing one or more environment variables, delimited by "%" characters, and returns the string with each environment variable replaced by its value. E.g. if the `PROMPT` environment variable has value 'pg' then

```
ExpandEnvVars('The prompt is %PROMPT%')
```

returns `'The prompt is $p$g'`.

```
function ExpandEnvVars(const Str: string): string;
var
  BufSize: Integer; // size of expanded string
begin
  // Get required buffer size
  BufSize := ExpandEnvironmentStrings(
    PChar(Str), nil, 0);
  if BufSize > 0 then
  begin
    // Read expanded string into result string
    SetLength(Result, BufSize - 1);
    ExpandEnvironmentStrings(PChar(Str),
      PChar(Result), BufSize);
  end
  else
    // Trying to expand empty string
    Result := '';
end;
```

Listing 5

This function can be useful when reading strings of type *REG_EXPAND_SZ* from the registry since these strings contain un-expanded environment variables. Passing such a string to this function will expand all the variables.

CreateEnvBlock

This final function creates an environment block that can be passed to a child process.

It creates a new environment block containing the strings from the *NewEnv* string list. If *IncludeCurrent* is true then the variables defined in the current process' environment block are included. The new block is stored in the memory pointed to by *Buffer*, which must be at least *BufSize* characters. The size of the block is returned. If the provided buffer is nil or is too small then no block is created. The return value gives the required buffer size in characters.

Be careful

Buffer sizes here are in characters, not bytes. On Unicode Delphis this is not the same thing.

Either allocate memory using *StrAlloc*, which allows for character size, or use *GetMem* and mulitply buffer size by *SizeOf(Char)*.

```
function CreateEnvBlock(const NewEnv: TStrings;
  const IncludeCurrent: Boolean;
  const Buffer: Pointer;
  const BufSize: Integer): Integer;
var
  EnvVars: TStringList; // env vars in new block
  Idx: Integer;         // loops thru env vars
  PBuf: PChar;          // start env var entry in block
begin
  // String list for new environment vars
  EnvVars := TStringList.Create;
  try
    // include current block if required
    if IncludeCurrent then
      GetAllEnvVars(EnvVars);
    // store given environment vars in list
    if Assigned(NewEnv) then
      EnvVars.AddStrings(NewEnv);
    // Calculate size of new environment block
    Result := 0;
    for Idx := 0 to Pred(EnvVars.Count) do
      Inc(Result, Length(EnvVars[Idx]) + 1);
    Inc(Result);
    // Create block if buffer large enough
    if (Buffer <> nil) and (BufSize >= Result) then
    begin
      // new environment blocks are always sorted
      EnvVars.Sorted := True;
      // do the copying
      PBuf := Buffer;
      for Idx := 0 to Pred(EnvVars.Count) do
      begin
        StrPCopy(PBuf, EnvVars[Idx]);
        Inc(PBuf, Length(EnvVars[Idx]) + 1);
      end;
      // terminate block with additional #0
      PBuf^ := #0;
    end;
  finally
    EnvVars.Free;
  end;
end;
```

Listing 6

The way we use this function is similar to the idiom used by many Windows API functions (such as *GetEnvironmentVariable*). We first call the function with a nil buffer to find the required buffer size, then call it again with a buffer of correct size to receive the data.

This routine can be used along with the Windows *CreateProcess* API function to spawn a new process with only one environment variable (`FOO=Bar`) as follows:

```
function ExecProg(const ProgName: string; EnvBlock: Pointer): Boolean;
  {Creates new process for given program passing any given environment block}
var
  SI: TStartupInfo;          // start up info
  PI: TProcessInformation;   // process info
  CreateFlags: DWORD;        // process creation flags
  SafeProgName: string;      // program name: safe for CreateProcessW
begin
  // Make ProgName parameter safe for passing to CreateProcessW
  SafeProgName := ProgName;
  UniqueString(SafeProgName);
  // Set up startup info record: all default values
  FillChar(SI, SizeOf(SI), 0);
  SI.cb := SizeOf(SI);
  // Set up creation flags: special flag required for unicode
  // environments, which is want when unicode support is enabled.
  // NOTE: we are assuming that the environment block is in Unicode
  // on Delphi 2009 or later. CreateProcessW does permits it to be
  // ANSI, but we don't support that
  {$IFDEF UNICODE}
  CreateFlags := CREATE_UNICODE_ENVIRONMENT;  // passing a unicode env
  {$ELSE}
  CreateFlags := 0;
  {$ENDIF}
  // Execute the program
  Result := CreateProcess(
    nil, PChar(SafeProgName), nil, nil, True,
        CreateFlags, EnvBlock, nil, SI, PI
  );
end;
```

Listing 7

Problems with *CreateProcess* Unicode API.

When using the Unicode API, *CreateProcess* maps to *CreateProcessW* which exhibits some strange behaviour. See *MSDN* for an explanation of the problem: look under the *lpCommandLine* parameter section.

We must make the *ProgName* parameter safe for passing to *CreateProcess* on Delphi 2009 and later by making sure the memory space occupied by *ProgName* is writeable. If *ExecProg* is called with a *ProgName* parameter with a reference count of -1 the memory is not writeable and *CreateProcess* will fail.

In a *post* to the CodeGear forums, Remy Lebeau suggested the *UniqueString* work-around that we have used. This ensures that *ProgName* has a reference count of 1.

Demo Code

A demo program to accompany this article is available for download. This program demonstrates the use of the six routines. Unzip the demo application and load it into Delphi to compile. A Readme file that explains how to use the demo program is included.

Download the demo code

Going Further

My *Environment Variables Unit* provides a set of routines and a component that use the methods discussed here to access environment variables.

Feedback

I hope you enjoyed this article and found it useful. If you have any observations, comments or have found any errors please *contact me*.