How to get operating system version information (part 3 of 6)

Extended OS information for the Windows NT platform

Extended Windows API OS Information

So far we can detect the name of an OS product, but for the NT platform that's not enough. Think for a moment about Windows XP – there are both the Home and Professional editions. For Windows 2000 it's similarly complicated – we have both server (e.g. Advanced Server) and workstation (Professional) editions.

TOSVersionInfoEx

So, how do we get to this information? We make a start by examining the information provided by an extension of *TOSVersionInfo* named, unsurprisingly, *TOSVersionInfoEx*. *Listing 10* shows a Pascal declaration of the structure. Note that Delphi does not declare it, so we need to include the definition in the interface section of `UOSInfo.pas`.

```
type
  TOSVersionInfoEx = packed record
    dwOSVersionInfoSize: DWORD;
    dwMajorVersion: DWORD;
    dwMinorVersion: DWORD;
    dwBuildNumber: DWORD;
    dwPlatformId: DWORD;
    szCSDVersion: array[0..127] of AnsiChar;
    wServicePackMajor: WORD;
    wServicePackMinor: WORD;
    wSuiteMask: WORD;
    wProductType: Byte;
    wReserved: Byte;
  end;
```

Listing 10

We can see that the first six fields of *TOSVersionInfoEx* are the same as *TOSVersionInfo* (explained in *Table 1*). The new structure simply adds fields to the end of the old structure. *Table 4* explains the purpose of the new fields:

Additional fields of TOSVersionInfoEx

Field	Description
wServicePackMajor	Major version number of the latest Service Pack installed on the system. If no Service Pack has been installed, the value is zero.
wServicePackMinor	Minor version number of the latest Service Pack installed on the system.
wSuiteMask	Bit mask that identifies the product suites available on the system. This member can be a combination of the numerous *VER_SUITE_** values defined by Microsoft. The values we will use here are: *VER_SUITE_ENTERPRISE* Windows Server 2003 Enterprise Edition, Windows 2000 Advanced Server or Windows NT 4.0 Enterprise Edition. *VER_SUITE_DATACENTER* Windows Server 2003 Datacenter Edition or Windows 2000 Datacenter Server. *VER_SUITE_PERSONAL* Windows XP Home Edition.

	VER_SUITE_BLADE Windows Server 2003 Web Edition. See Microsoft's documentation for a description of the other available values.
wProductType	Additional information about the operating system product. This member can take one of the following values: *VER_NT_WORKSTATION* The system is a workstation edition. It is running Windows XP Professional, Windows XP Home, Windows 2000 Professional or Windows NT 4.0 Workstation. *VER_NT_DOMAIN_CONTROLLER* The system is a domain controller. *VER_NT_SERVER* The system is a server.
wReserved	Reserved for future use.

Table 4

Delphi defines neither the *VER_NT_** nor *VER_SUITE_** constants discussed in *Table 4*, so we must add the definitions to our `UOSInfo` unit's interface section. *Listing 11* shows the required definitions. Note that the listing declares all documented *VER_SUITE_** flags, not just those described in *Table 4*.

```
const
  // NT Product types: used by dwProductType field
  VER_NT_WORKSTATION = $0000001;
  VER_NT_DOMAIN_CONTROLLER = $0000002;
  VER_NT_SERVER = $0000003;

  // NT product suite mask values: used by wSuiteMask field
  VER_SUITE_SMALLBUSINESS = $00000001;
  VER_SUITE_ENTERPRISE = $00000002;
  VER_SUITE_BACKOFFICE = $00000004;
  VER_SUITE_COMMUNICATIONS = $00000008;
  VER_SUITE_TERMINAL = $00000010;
  VER_SUITE_SMALLBUSINESS_RESTRICTED = $00000020;
  VER_SUITE_EMBEDDEDNT = $00000040;
  VER_SUITE_DATACENTER = $00000080;
  VER_SUITE_SINGLEUSERTS = $00000100;
  VER_SUITE_PERSONAL = $00000200;
  VER_SUITE_SERVERAPPLIANCE = $00000400;
  VER_SUITE_BLADE = VER_SUITE_SERVERAPPLIANCE;
```

Listing 11

So, we've got a new structure that gives us extra information to play with, but how to we get the information from Windows?

Populating TOSVersionInfoEx

TOSVersionInfoEx is only supported on some NT platforms and not at all on the Windows 9x platform. As a result we have to check that we can use it. Here's an apparent paradox: how do we check what OS we are using before calling the code that checks the OS? The answer is quite simple, if a little convoluted.

NT support for *TOSVersionInfoEx*

TOSVersionInfoEx is supported on all recent NT products: i.e. all those except NT 4 Service Pack 5 and earlier.

We call *GetVersionEx()* passing it a *TOSVersionInfoEx* instead of a *TOSVersionInfo* structure. If that call fails we have to pass *GetVersionEx()* a *TOSVersionInfo* structure instead. In actual fact we pass *TOSVersionInfoEx* again with the *dwOSVersionInfoSize* field set to the size of a *TOSVersionInfo* structure. This works because the first fields of *TOSVersionInfoEx* are the same as *TOSVersionInfo*.

Listing 12 shows how this is done. On OSs that do not support *TOSVersionInfoEx*, only the first six fields will be completed. The *IsExtended* variable is true if the whole of the structure is populated and false if not.

```
var
  POSV: POSVersionInfo; // pointer to OS version info structure
  IsExtended: Boolean;  // flags whether extended structure used
```

```
begin
  ...
  // Clear the structure
  FillChar(OSV, SizeOf(OSV), 0);
  // Get pointer to structure of non-extended type
  // (GetVersionEx requires non-extended structure: we need
  // this pointer to get it to accept our extended structure!!)
  {$TYPEDADDRESS OFF}
  POSVI := @OSV;
  {$TYPEDADDRESS ON}
  // We first try to get extended information
  OSV.dwOSVersionInfoSize := SizeOf(TOSVersionInfoEx);
  IsExtended := GetVersionEx(POSV^);
  if not IsExtended then
  begin
    // We failed to get extended info: try with old structure
    OSV.dwOSVersionInfoSize := SizeOf(TOSVersionInfo);
    if not GetVersionEx(POSV^) then
      // We failed again: shouldn't happen so raise exception
      raise Exception.Create('Can''t get OS info');
  end;
  ...
end;
```

Listing 12

Let us examine the code in detail. We first zero the *TOSVersionInfoEx* record then set its *dwOSVersionInfoSize* field to the size of the extended structure. Then we call the *GetVersionEx()* API function. If this function fails we reset the *dwOSVersionInfoSize* field to the size of a *TOSVersionInfo* structure and try again. This call should succeed. If it fails we raise an exception.

Now the `Windows` unit defines *GetVersionEx()* to accept a *TOSVersionInfo* parameter rather than one of type *TOSVersionInfoEx*. Because of Delphi's strong typing we have to be a little underhand to get *GetVersionEx()* to accept our *TOSVersionInfoEx* parameter. This is accomplished by taking the address of the *TOSVersionInfoEx* record and casting it to a pointer to a *TOSVersionInfo* structure. Then we de-reference the pointer when passing to *GetVersionEx()*. Dirty but it works!

In the above code the *IsExtended* variable enables us to remember whether the structure contains extended information or not. Another method of checking is to read the structure size stored in the *dwOSVersionInfoSize* field – it will be *SizeOf(TOSVersionInfoEx)* if we have extended information and *SizeOf(TOSVersionInfo)* if not.

We now have some additional information at our disposal with which to describe later NT platform operating systems. Rather than using this directly, we will follow Delphi's example and provide some global variables to store the extended OS version information. Later we will adapt our *TOSInfo* class to use these new global variables.

Extending Delphi's RTL Support

Since Delphi provides global variables that it sets during start-up to the values of the fields of *TOSVersionInfo* it makes sense to provide more global variables that echo the fields of *TOSVersionInfoEx*. We will also provide a Boolean variable that records whether the extended information is available. The new variables are described in *Table 5*.

Additional Win32XXX global variables

Variable	Field	Notes
Win32ServicePackMajor	*wServicePackMajor*	Major version of any installed service pack or 0 if no such pack. Default value 0.
Win32ServicePackMinor	*wServicePackMinor*	Minor version of any installed service pack. Default value 0.
Win32SuiteMask	*wSuiteMask*	Bit flags that identify the product suites available on the system. Valid bit flags are defined by the *VER_SUITE_XXX* constants†. Default value 0.

Win32ProductType	*wProductType*	Additional information about the operating system. Possible values are given by the *VER_NT_XXX* constants†. Default value 0.
Win32HaveExInfo	– N/a –	Flag true if we have extended operation system version information and false if not. When this flag is false the variables above have no meaning and are zeroed.

Table 5

† The *VER_SUITE_XXX* and *VER_NT_XXX* constants are defined in *Listing 11* and described in *Table 4*.

We will declare the new global variables in the interface section of `UOSInfo` as shown in *Listing 13*.

```
var
  Win32HaveExInfo: Boolean = False;
  Win32ServicePackMajor: Integer = 0;
  Win32ServicePackMinor: Integer = 0;
  Win32SuiteMask: Integer = 0;
  Win32ProductType: Integer = 0;
```

Listing 13

All that remains to do is to try to get the extended information at start up and to store the required values in the global variables. Using the `SysUtils` *InitPlatformId* procedure as an example we will add a routine named *InitPlatfornIdEx* to `UOSInfo`'s implementation section and call it in the initialization section. *Listing 14* gives the required code.

```
procedure InitPlatformIdEx;
var
  OSVI: TOSVersionInfoEx;
  POSVI: POSVersionInfo;
begin
  FillChar(OSVI, SizeOf(OSVI), 0);
  {$TYPEDADDRESS OFF}
  POSVI := @OSVI;
  {$TYPEDADDRESS ON}
  OSVI.dwOSVersionInfoSize := SizeOf(TOSVersionInfoEx);
  Win32HaveExInfo := GetVersionEx(POSVI^);
  if Win32HaveExInfo then
  begin
    Win32ServicePackMajor := OSVI.wServicePackMajor;
    Win32ServicePackMinor := OSVI.wServicePackMinor;
    Win32SuiteMask := OSVI.wSuiteMask;
    Win32ProductType := OSVI.wProductType;
  end;
end;

...

initialization

InitPlatformIdEx;

end.
```

Listing 14

This routine works in a similar same way to that presented in *Listing 12*. The main difference is that we don't try to call *GetVersionEx* with *TOSVersionInfo* if the call with *TOSVersionInfoEx* fails. This is not necessary since Delphi has already recorded the information that *TOSVersionInfo* provides. We simply use *Win32HaveExInfo* to record whether *GetVersionEx* succeeds and set the remaining global variables if so.

OS Version Information Class: Version 2

Now that we've explored how to get the additional OS information from the Windows API we are ready to extend our OS version information class accordingly.

Service Pack Versions

Unsupported NT OSs

These methods do not work for NT4 service pack 5 and earlier – the *ServicePack* method should be checked in these cases.

We will begin by adding two new parameterless class methods – *Win32ServicePackMajor* and *Win32ServicePackMinor* – and implementing them as shown in *Listing 15*. The methods return the major and minor service pack versions for supported Windows NT operating systems or 0 if the OS is not supported or has no service pack applied. As can be seen in the listing, the methods simply return the value of the corresponding global variable that we defined in the previous section.

```
class function TOSInfo.ServicePackMajor: Integer;
begin
  Result := Win32ServicePackMajor;
end;

class function TOSInfo.ServicePackMinor: Integer;
begin
  Result := Win32ServicePackMinor;
end;
```

Listing 15

Product Type

Now we turn our attention to the information stored in the *Win32ProductType* variable. We have the following possibilities:

- The variable is not valid, either because we have an NT platform OS that doesn't support the extended OS info, or because we have a non-NT platform.
- We have a supported NT platform in which case the variable informs us whether we have a workstation, server or domain controller system.

We will define a class method to distinguish these possibilities. The method will return an enumerated value of type *TOSProductType* that we will define in our unit's interface section as:

```
type
  TOSProductType = (
    ptNA,                   // not applicable: not a Windows NT platform
    ptUnknown,              // unknown NT product type
    ptNTWorkstation,        // NT workstation
    ptNTServer,             // NT server
    ptNTDomainController    // NT domain controller
  );
```

Listing 16

Listing 17 shows the implementation of the new method, which should have its prototype added to the class declaration.

```
class function TOSInfo.ProductType: TOSProductType;
begin
  if IsWinNT then
  begin
    case Win32ProductType of
      VER_NT_WORKSTATION: Result := ptNTWorkstation;
      VER_NT_SERVER: Result := ptNTServer;
      VER_NT_DOMAIN_CONTROLLER: Result := ptNTDomainController;
      else Result := ptUnknown;
    end;
  end
  else
    Result := ptNA;
end;
```

Listing 17

We first check to see we have an NT platform OS and return *ptNA* if not. If we have an NT platform OS then we check the value of the *Win32ProductType* global against its possible values and return the associated value from *TOSProductType* if a match is found. Where the value is unrecognised we return *ptUnknown*. You may have noticed that we don't use *Win32HaveExInfo* to check whether extended OS information is supported. This is because *Win32ProductType* is set to zero in this event, causing all the **case** clauses to fail and *ptUnknown* to be returned.

Checking for Server OSs

We may also find it useful to be able to quickly check whether the operating system is a server system. To do this we add a new class method named *IsServer* to *TOSInfo*. *Listing 18* defines the method.

```
class function TOSInfo.IsServer: Boolean;
begin
  Result := ProductType in [ptNTServer, ptNTDomainController];
end;
```

Listing 18

The method is defined in terms of the *ProductType* method – it simply checks if *ProductType* returns a value that represents an NT server OS.

Warning

IsServer will always return false if we have an NT4 server that has service pack 5 or lower. The solution of this problem will be discussed *later in the article*.

Finding the OS Edition

As we have already noted, each NT operating system version is sub-divided into various "editions". We will now extend *TOSInfo* to be able to detect and describe these editions. We do this by defining a new class method – *Edition* – that returns a string that describes the edition. The method will return the empty string if the edition is not known or if we are not using the NT platform. *Listing 19* shows the method's implementation.

```
class function TOSInfo.Edition: string;
begin
  Result := '';
  if IsWinNT then
  begin
    if Win32HaveExInfo then
    begin
      // Test for edition on Windows NT 4 SP6 and later
      if IsServer then
      begin
        // Server type
        case Product of
          osWinNT4:
          begin
            if CheckSuite(VER_SUITE_ENTERPRISE) then
              Result := 'Server 4.0, Enterprise Edition'
            else
              Result := 'Server 4.0';
          end;
          osWin2000:
          begin
            if CheckSuite(VER_SUITE_DATACENTER) then
              Result := 'Datacenter Server'
            else if CheckSuite(VER_SUITE_ENTERPRISE) then
              Result := 'Advanced Server'
            else
              Result := 'Standard Edition';
          end;
          osWinServer2003:
          begin
            if CheckSuite(VER_SUITE_DATACENTER) then
              Result := 'DataCenter Edition'
            else if CheckSuite(VER_SUITE_ENTERPRISE) then
            Result := 'Enterprise Edition'
          else if CheckSuite(VER_SUITE_BLADE) then
            Result := 'Web Edition'
          else
            Result := 'Standard Edition';
        end;
      end;
    end
    else
    begin
      // Workstation type
      case Product of
        osWinNT4:
          Result := 'Workstation 4.0';
        osWin2000:
          Result := 'Professional';
        osWinXP:
        begin
          if CheckSuite(VER_SUITE_PERSONAL) then
            Result := 'Home Edition'
          else
            Result := 'Professional';
        end;
      end;
    end;
  end;
 end;
end;
```

Listing 19

Unsupported NT OSs

Once again, this method will not work for NT4 service pack 5 and earlier. An empty string will be returned by *Edition* in this case. We will fix this problem *later in the article* where we will modify the method to work when *Win32HaveExInfo* is false.

We first set the result to the empty string in case we fail to find any information about the edition. We then check that we have an NT operating system – we are done if not. Next we check that we have extended OS information available by examining *Win32HaveExInfo*. Again we having nothing more to do if this variable is false.

The rest of the method depends on whether we have a server OS or a workstation version. We proceed similarly in either case. First we determine the OS product. For some workstations this is all the information we need to determine the edition (for example the only Windows 2000 workstation is the Professional Edition). In other cases we then need to check the operating system suite bitmask (stored in *Win32SuiteMask*) to see which *VER_SUITE_** constants it contains. This bitmask gives us the appropriate edition.

To help keep *Edition*'s code tidy we define a private helper method – *CheckSuite* – to examine the *Win32SuiteMask* bitmask. We pass *CheckSuite* the required bit flag and it returns true if *Win32SuiteMask* contains the flag. *CheckSuite* is defined in *Listing 20*.

```
class function TOSInfo.CheckSuite(const Suite: Integer): Boolean;
begin
  Result := Win32SuiteMask and Suite <> 0;
end;
```

Listing 20

This completes our discussion of extended OS information available from Windows. In the *next section* we will look at getting more information about early NT 4 operating systems from the registry.