Delphi Internals: CPU Type

Component-based CPU Type determination

by Dave Jewell

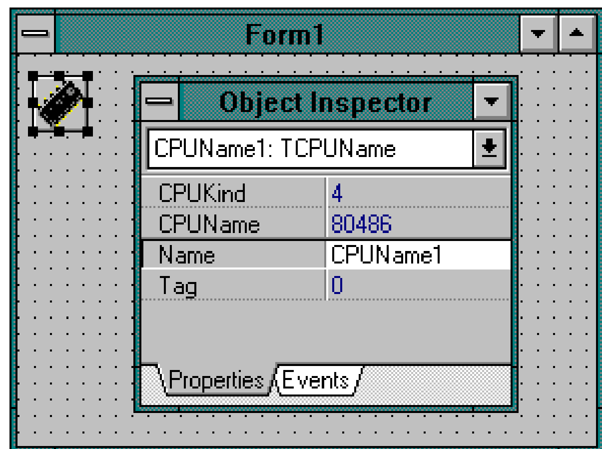
A few weeks ago, I was given the task of writing a readership survey program for a popular UK magazine. Not unnaturally, I decided to write the program in Delphi! Much of the required information was provided by getting the user to answer a series of questions, but it was also important to automatically gather some information on the hardware which the user was running. One item that was of interest to the publishers was the CPU type.

Initially, I planned to use the `GetWinFlags` API routine to determine what sort of processor was in use. However, I discovered that this routine doesn't even recognise Pentium processors let alone the Pentium Pro (P6) chip. It seemed to me that this wasn't adequate for the job so I decided to look further afield. Somewhat later, I discovered a call you could make on the built-in Windows DPMI server which will return the CPU type. This call is accessed by a `INT $31` call. Again, this looked promising, but as before there was no information on how to recognise Pentium or Pentium Pro chips.

In desperation I started trawling the bulletin boards and eventually found what I was looking for in the Delphi forum on CompuServe. I discovered some code (kindly donated by Intel) which reliably checks for all existing processor types – oh joy!

Introducing The TCPUName Component

As it stood, the code in question was rather untidy and mainly comprised a single large in-line assembler routine. I spent some time tidying this up and decided to re-package it as a component. One might as well do things properly! The screenshot above shows how the component looks from the viewpoint of the Object Inspector.



➤ *Our TCPUName component in action! The corresponding Object Inspector window is also shown*

The component has two special properties in addition to the `Name` and `Tag` fields. Both of these properties are read-only. If you try editing them in the Object Inspector, they'll immediately snap back to whatever values they had beforehand. More on that shortly!

The first property, `CPUKind`, is a simple integer value which returns the type of CPU we're dealing with. This number will generally take one of the following values:

```
const
  i8086       = 1; {also 8088}
  i80286      = 2;
  i80386      = 3;
  i80486      = 4;
  iPentium    = 5;
  iPentiumPro = 6;
```

Although the constants defined here only go up as far as the Pentium Pro, you can reliably expect that what comes after will return a value of 7, provided that Intel continue to consistently implement the `CPUID` instruction which is used by the `TCPUName` code. `CPUID`, in case you haven't encountered it before, is a special instruction which returns the type of processor being used along with other assorted information. Unfortunately Intel didn't think of implementing the `CPUID` instruction until they got as far as the 80486 processor (amazing how obvious things seem given the benefit of hindsight!) which is why the code in Listing 1 is a good deal more complex than it would otherwise have to be.

The second property, `CPUName`, returns a plain-English description of the processor. This is for use by those applications (like my reader survey program) which simply wish to report the CPU type without necessarily doing anything else with it. The possible values which this string can take are:

```
'8086'
'80286'
'80386'
'80486'
'Pentium'
'Pentium Pro'
'Px'
```

The last item here is intended for upward compatibility with future processors. Although we can't predict Intel's marketing names for each processor, the 'Px' designation should be consistent. I've already got my order in for a quad, 2 GHz P10 motherboard...

How It Works

Listing 1 shows the component in its entirety. Two private member functions are implemented within the class definition, `GetCPUKind` and

GetCPUName. These methods are used to return the processor type and processor name respectively for the implementation of the two properties described above. It's likely that you'll only ever have one instance of the TCPUName component in any one application, but for the sake of simplicity, and to avoid re-interrogating the CPU where multiple instances might be present, I decided to store the CPU identifier using a single integer global variable. This is set up in the unit initialisation code.

You'll also notice that there are two dummy routines, NOPInteger and NOPString. These are used to provide dummy write methods for the two component properties. The recommended way of creating read-only properties (according to the Delphi on-line help) is to declare properties without an associated write clause, like this:

```
published
  { Published declarations }
  property CPUKind: Integer
    read GetCPUKind; {read-only!}
  property CPUName: String
    read GetCPUName; {read-only!}
end;
```

Listing 1

```
unit CPUKind;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs;
type
  TCPUName = class(TComponent)
    private
      { Private declarations }
      function GetCPUKind: Integer;
      function GetCPUName: String;
      procedure NOPInteger (val: Integer);
      procedure NOPString (val: String);
    protected    { Protected declarations }
    public       { Public declarations }
    published
      { Published declarations }
      property CPUKind: Integer read GetCPUKind
        write NOPInteger; { read-only! }
      property CPUName: String read GetCPUName
        write NOPString;   { read-only! }
  end;
procedure Register;
implementation
const
  i8086       = 1;            { includes 8088 CPU as well }
  i80286      = 2;
  i80386      = 3;
  i80486      = 4;
  iPentium    = 5;            { P5 - Pentium }
  iPentiumPro = 6;            { P6 - Pentium Pro }
var
  id: Integer;
function CpuID: Integer; assembler;
{ Assembly function to get CPU type incl Pentium and later }
asm
  push        ds              { first, check for 8086 -
                                Flag bits 12-15 always set }
  call        GetWinFlags     { call Windows API }
  or          ax,wf_CPU286    { or with 80286 processor bit }
  mov         ax,i80286       { assume 286 }
  jz          @@1             { branch if it was }
  { Not a 80286 - let's check for a 8088/8086 next }
  pushf                       { save EFLAGS }
  pop         bx              { store EFLAGS in BX }
  mov         ax,0fffh        { clear bits 12-15 }
  and         ax,bx           { in EFLAGS }
  push        ax           { store new EFLAGS value on stack }
  popf                        { replace current EFLAGS value}
  pushf                       { set new EFLAGS }
  pop         ax              { store new EFLAGS in AX }
  and         ax,0f000h   { if bits 12-15 are set, then 8086 }
  cmp         ax,0f000h       { is an 8086/8088 ? }
  mov         ax,i8086        { turn on 8086/8088 flag }
  je          @@1             { yes - all done }
  { To test for 386 or better, we need to use 32 bit
    instructions, but the 16-bit Delphi assembler does not
    recognize the 32 bit opcodes or operands.  Instead, use
    the 66H operand size prefix to change each instruction to
    its 32-bit equivalent. For 32-bit immediate operands, we
    also need to store the high word of the operand
    immediately following the instruction. The 32-bit
    instruction is shown in a comment after the 66H
    instruction. }
  db          66h             { pushfd }
  pushf
  db          66h             { pop eax }
  pop         ax              { get original EFLAGS }
  db          66h             { mov ecx, eax }
  mov         cx,ax           { save original EFLAGS }
  db          66h             { xor eax,40000h }
  xor         ax,0h           { flip AC bit in EFLAGS }
  dw          0004h
  db          66h             { push eax }
  push        ax              { save for EFLAGS }
  db          66h             { popfd }
  popf                        { copy to EFLAGS }
  db          66h             { pushfd }
  pushf                       { push EFLAGS }
  db          66h             { pop eax }
  pop         ax              { get new EFLAGS value }
  db          66h             { xor eax,ecx }
  xor         ax,cx        { can't toggle AC bit, CPU=Intel386 }
  mov         ax,i80386       { turn on 386 flag }
  je          @@1
  { i486 DX CPU / i487 SX MCP and i486 SX CPU checking
    Checking for ability to set/clear ID flag (Bit 21) in
    EFLAGS which indicates the presence of a processor with
    the ability to use the CPUID instruction }
  db          66h             { pushfd }
  pushf                       { push original EFLAGS }
  db          66h             { pop eax }
  pop         ax              { get original EFLAGS in eax }
  db          66h             { mov ecx, eax }

  mov         cx,ax           { save original EFLAGS in ecx }
  db          66h             { xor eax,200000h }
  xor         ax,0h           { flip ID bit in EFLAGS }
  dw          0020h
  db          66h             { push eax }
  push        ax              { save for EFLAGS }
  db          66h             { popfd }
  popf                        { copy to EFLAGS }
  db          66h             { pushfd }
  pushf                       { push EFLAGS }
  db          66h             { pop eax }
  pop         ax              { get new EFLAGS value }
  db          66h             { xor eax, ecx }
  xor         ax, cx
  mov         ax,i80486       { turn on i486 flag }
  je          @@1
  { if ID bit cannot be changed, CPU=486 without CPUID
    instruction functionality }
  { Execute CPUID instruction to determine vendor, family,
    model and stepping.  The CPUID instruction used in this
    program can be used for B0 and later steppings of P5 }
  db          66h             { mov eax, 1 }
  mov         ax, 1           { set up for CPUID instruction }
  dw          0
  db          66h             { cpuid }
  db          0Fh     { Hardcoded opcode for CPUID instruction }
  db          0a2h
  db          66h             { and eax, 0F00H }
  and         ax, 0F00H       { mask everything but family }
  dw          0
  db          66h             { shr eax, 8 }
  shr         ax, 8     { shift the cpu type down to low byte }
@1:
  pop         ds
end;
procedure TCPUName.NOPInteger(val: Integer); begin end;
procedure TCPUName.NOPString(val: String); begin end;
function TCPUName.GetCPUKind: Integer;
begin
  Result := id;
end;
function TCPUName.GetCPUName: String;
begin
  case id of
    i8086:       Result := '8086';
    i80286:      Result := '80286';
    i80386:      Result := '80386';
    i80486:      Result := '80486';
    iPentium:    Result := 'Pentium';
    iPentiumPro: Result := 'Pentium Pro';
  else
    Result := Format ('P%d', [id]);
  end;
end;
procedure Register;
begin
  RegisterComponents ('Pilgrim''s Progress', [TCPUName]);
end;
begin
  id := CpuID;        { unit initialisation }
end.
```

Unfortunately, if you try this, you'll find that not only are the properties read-only, but they are also invisible to the Object Inspector. I found this rather irritating as I wanted to see the properties in the Object Inspector window. In order to get the read-only properties to appear, it was necessary to fool Delphi into thinking that the properties were writeable, hence the need for the dummy write methods. I think this is a shortcoming. After all, it's called an Object *Inspector*, so you would imagine that it ought to be able to *inspect* read-only properties without any implication that the properties in question are writeable! Ho-hum...

The most important routine in Listing 1 is the `CpuID` function. This first calls the `GetWinFlags` API function in order to determine if a 286 is in use. Although the original Intel source included 286 detection code, this was commented out and I got the impression it caused GPF problems under Windows.

The program then tries to see if it's dealing with an 8088/8086 by checking to see if certain bits are 'stuck' in the `EFLAGS` register. If you're paying attention, you'll know that there's a zero percent chance of detecting a 8088/8086 processor while running a Delphi program since these processors aren't even capable of entering protected mode, let alone running Windows! Nevertheless, I've left the code in so that you can adapt it for use in a DOS application if you wish. The same argument applies to 286 detection. Most modern software, including Windows 95 itself, requires a minimum of a 386 processor, but again, I've left the code in for the sake of completeness.

From then onwards, the code uses 32-bit instructions to test for 386 (and higher) processors. Unfortunately, the in-line assembler built into 16-bit Delphi won't recognise anything higher than 286 instructions, so we have to manually prefix each 32-bit instruction with the value `$66`. This looks rather untidy, but the result is the same. Each `$66` op-code tells the processor to treat the next instruction as being 32-bit rather than 16. Thus, the op-code `$50`, which is interpreted as `PUSH AX`, will normally only push the contents of the 16-bit `AX` register onto the stack. However, if we precede the `$50` op-code with `$66`, then it becomes a `PUSH EAX` instruction, pushing the entire contents of the 32-bit `EAX` register.

Incidentally, 32-bit Delphi will be able to assemble 32-bit instructions directly, so if you want to port this code over you'll be able to significantly tidy it up and get rid of the 8088/8086 and 286 checks at the same time!

By the time the code has got to this point, we know that it's a 486 processor (or better). However, not all 486 processors implement the `CPUID` instruction which returns the processor family and other information. Accordingly, the code has to perform another check, testing bit 21 in the `EFLAGS` register to see if `CPUID` functionality is present. If it isn't, we know it's just a low-end 486 chip. Otherwise the code executes a `CPUID` instruction and returns the family identifier directly as the function result.

The CPUID Instruction

If you wanted to, you could easily obtain more detailed information from the call to `CPUID`. If you look at Table 3 on page 4 of the Intel PDF file on the disk (see *'What's On The Disk'*), you'll see a description of the format of the `EAX` register immediately after `CPUID` has been executed. This gives you the minor stepping number, major stepping number, family information (which is what is picked up by the `TCPUName` component) and model number. Using this information, you could easily add a model number and/or string to the `TCPUName` property list, and the same for the stepping information. This is left as an exercise for the reader, but it's clearly very straightforward.

Getting the model and stepping information might be advantageous in a hardware diagnostic program, but bear in mind that if you're seeking to detect the notorious Pentium division bug you won't get very far because Intel didn't change the processor stepping numbers when they fixed the divide instruction. Instead, you actually have to perform some division and see if it checks out OK.

After the `CPUID` instruction has executed, the `EDX` register contains a list of 'feature flags' for the processor. The format of these flags is given in the Intel PDF file, although I think the information here is a bit too esoteric to be of much general use. I can't help wondering, though, if the undocumented feature flags perhaps provide an easy way to detect the fixed division problem, amongst other things.

What's On The Disk

The disk with this issue contains the component files as well as the Intel documentation on the `CPUID` instruction, in Adobe Acrobat format as CPUAP.PDF (the Acrobat reader is on your Delphi CD).

Dave Jewell is a freelance technical journalist, computer consultant and author of *Instant Delphi* from Wrox Press. This article is based on part of his forthcoming book on Delphi component writing, which will be published in the first half of 1996. You can reach Dave by email on the internet as djewell@cix.compulink.co.uk or on CompuServe as 102354,1572