Typecasting Explained: Part 2

by Brian Long

In the last issue, I was talking about common uses for typecasting, including dissecting pointers. Before the advent of Delphi, and indeed since then for those who haven't come across the better alternatives, users of Borland's Pascal-based products dealt with Windows messages just as with any other third generation language.

All Windows messages were sent in a particular fixed message structure (a `TMessage` record, usually called `Message`) regardless of the format of the additional information they were intended to convey.

The `TMessage` structure is defined in the *Component Writer's Guide* help file and principally holds the message number (`Msg`), a word parameter (`WParam`), a double-word parameter (`LParam`) and a result field (`Result`). Given the diversity of messages available and the information that needs to be imparted, two number fields are not sufficient to provide it in readily accessible format. Instead, you need to use typecasting to get the right bits out. Moreover, since there are so many messages, you need to have a photographic memory, or keep referring to some reference, to work out which bits go where. And in addition, in Win32, the `Word` and `Longint` change to two `Longints` with information packaged differently. What a mess.

Fortunately for us, Microsoft took the initiative and introduced message crackers into the Windows SDK. These were originally implemented as C macros and surface in Delphi as alternative message record structures, to be found in the `Messages` unit. There is a record type for each message. The point of message crackers is to provide easier access to the information being sent with a message and to allow platform-independent message-handling code to be written – instead of your code changing between platforms, the definitions of the message cracker records change. Let's look at an example, the `wm_GetMinMaxInfo` message. We can use this message, via a message handler for example, to modify the minimum and maximum size of a form, or any other window for that matter.

The information passed along with the message is a record of type `TMinMaxInfo`, which has five `TPoint` fields, where a `TPoint` is a record with two `Integer` fields, which makes a total of ten bytes. Obviously that's not going to fit into a `TMessage` record, so the address of this record is passed in the `Longint` field. To access the fields of the record, we need to apply a variable typecast to `Message.LParam` to make it look like a pointer to a `TMinMaxInfo` record, or a `PMinMaxInfo`, and then dereference the resultant pointer. So we use:

```
PMinMaxInfo(Message.LParam)^.ptMaxSize :=
  PointVariable;
```

This typecasting of message fields used to be very common. With message crackers we define `Message` to be a `TWMGetMinMaxInfo` record rather than a `TMessage`, and use:

```
Message.MinMaxInfo^.ptMaxSize :=
  PointVariable;
```

Unsafe And Safe Typecasting Of Objects

Having smartly avoided objects all this time, let's turn and face them head-on. You have probably noticed that all the event handlers for any components on a form are implemented as methods of that form. This is called delegation.

The objects are said to delegate the handling of the events to another object. Since there is already a new form class definition being built up by the form designer, with data fields for each added component, the form is the component that gets to hold all the event handlers.

This is very much unlike C++ and previous Pascal models, where an event handler for an object would live in that object's class definition (after having derived a new class to define the event handler to be part of). There are two clear advantages to the delegation model. Firstly, if there are less class definitions around, there is less code required for a program to work. But more importantly, if most objects are delegating their event handling to the form, we can share event handlers between objects. One event handler can be triggered from more than one place, perhaps more than one object. This begs the question of "How do we know which object triggered it?" That's what the `Sender` parameter is for in practically all of these event handlers the VCL provides.

Take an example where there are two buttons side by side on a form, `Button1` and `Button2`. Double click the first button to generate an `OnClick` handler for it. Now go back to the form designer (press F12), select the other button (right cursor key) and go to the Events page of the Object Inspector (press F11, Ctrl-Tab). Drop down the list of candidate event handlers for the `OnClick` event (Alt-down cursor key) and choose the event handler just generated, `Button1Click`. That has associated the one event handler with an event in two distinct components.

Okay, now suppose we wish to put some code in the handler to change the caption of the button which caused the event to trigger. That's where `Sender` comes in. `Sender` is declared as `TObject`, a sort of lowest common denominator class, and because a `TObject` does not have anything in it called `Caption`, the compiler will reject something like this, despite the fact that we know `Sender` will be a `TButton` and so the sentiment is valid:

```
Sender.Caption := 'Clicked';
```

Instead, we need to take account of the fact that Pascal is strongly typed and suggest that the compiler considers Sender to be not of type TObject, but of type TButton. We can use a variable typecast to achieve this:

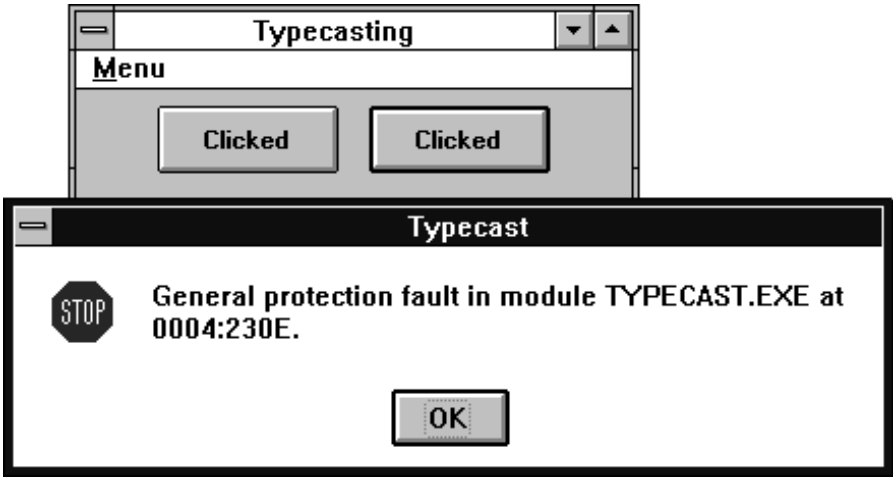
```
TButton(Sender).Caption :=
  'Clicked';
```

and indeed this works. You may recall one of the rules for a variable typecast was that the subject of the typecast was the same size in bytes as the target type.

The reason this typecast works, even though the TObject class will be rather smaller than the TButton class, is that all these declarations like Sender of type TObject, Button1 of type TButton etc, are declarations of object references not actual objects. An object reference is a pointer to an object, but without the complications of standard pointer syntax. This means objects are allocated on the heap by default, which is the largest memory store under Windows, and means they can be de-referenced easily. However, it does lead to a confusing inconsistency in the language which trips people up on a regular basis. But at any rate, any object reference will be the same size as any other, since all pointers are the same size: four bytes. This can lead to problems.

Let's continue the example. Add a main menu onto the form and define an item on the menu bar with a caption of Menu and a drop down item from it captioned Click. Select the Click menu item. Go to the Events page of the Object Inspector, drop down the list of candidate OnClick event handlers and pick Button1Click. Now run the program and choose the menu item. ***Boom!*** The debris is shown in Figure 1 in the shape of a GPF.

The GPF is caused by us telling the compiler to treat the object that caused the event as a button, when in fact it was a menu item. The trouble is, we might not even get a GPF. We only got one here because at some point an attempt was made to access memory which wasn't ours to access. We may just get some innocent data elsewhere being written over, causing untraceable errors later in the lifetime of the program.



➤ *Figure 1 So Delphi **can** do what the competition does so well...*

```
unit Typecstu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    MainMenu1: TMainMenu;
    Menu1: TMenuItem;
    Item1: TMenuItem;
    procedure GenericClick(Sender: TObject);
  private   { Private declarations }
  public    { Public declarations }
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.GenericClick(Sender: TObject);
begin
  {TButton(Sender).Caption := 'Clicked';}
  if Sender is TButton then
    (Sender as TButton).Caption := 'Clicked';
  if Sender is TMenuItem then
    (Sender as TMenuItem).Enabled := False;
  MessageBeep($FFFF);
end;
end.
```

➤ *Listing 1*

Unfortunately, the compiler can't warn us of the impending disaster because one pointer looks much like another to it. But there is a way we can avoid the problem. If we call this current form of variable typecasting *unsafe* in the context of objects, then we can also do *safe* typecasting using a different syntax, taking advantage of the reserved words that use run-time type information. A more appropriate version of the above typecast would be:

```
(Sender as TButton).Caption :=
  'Clicked';
```

which only performs a typecast if the type of Sender is TButton or any object derived from type TButton. If Sender is some other type, such as a TMenuItem, we still get an exception, but we will always get the exception, and the exception is an EInvalidCast exception which can be trapped for. Of course, it would be better to not get an exception at all and so we can employ the is keyword, as shown in Listing 1. The typecast is then only performed if the preceding test evaluates to True, ie it is going to work.

Epilogue: Is Pascal Less Flexible Than Assembler?

Who knew that Delphi, like Borland Pascal before it and Turbo Pascal before that, supports direct entry of inline assembly instructions, in addition to direct entry of machine code hex bytes?

It's alright. You can come out from under the table now. I'll be administering it in small, carefully measured doses...

Listing 2 shows an application with a reasonably simple main form. When the main form activates, its `OnActivate` event handler kicks in and starts a Windows-friendly loop that only finishes when the user closes the program down. Each time round the loop it updates a label on the form with the new value of a property of the form called `BIOSCounter`. `BIOSCounter`, when read, should return the current value of the PC BIOS timer tick counter. This double word value, stored in the lower recesses of DOS memory, is incremented every time the BIOS timer ticks, ie every 55ms.

The `BIOSCounter` property is a read-only property, which calls the `GetBIOSCounter` function when it is read. There are four candidate implementations of `GetBIOSCounter` in the listing that we will look at. Each one stores the current value, read directly from the BIOS Data Area, in a private data field of the

➤ *Listing 2*

```
unit Basmu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls,Forms, Dialogs, ExtCtrls, StdCtrls;
type
  TBIOSCounterForm = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    procedure FormActivate(Sender: TObject);
  private
    { Stores a copy of the BIOS timer counter }
    FBIOSCounter: Longint;
    { Updates FBIOSCounter }
    function GetBIOSCounter: Longint;
  public
    { Simple property to set up and return FBIOSCounter }
    property BIOSCounter: Longint read GetBIOSCounter;
  end;
var BIOSCounterForm: TBIOSCounterForm;

implementation
{$R *.DFM}
const
  { You really shouldn't do this... }
  BIOSArea = $40;
  BIOSCounterLo = $6C;
  BIOSCounterHi = $6E;
{$define VERSION1}

{$ifdef VERSION1}
function TBIOSCounterForm.GetBIOSCounter: Longint;
begin
  FBIOSCounter := MemL[BIOSArea:BIOSCounterLo];
  Result := FBIOSCounter;
end;
{$endif}

{$ifdef VERSION2}
function TBIOSCounterForm.GetBIOSCounter: Longint;
begin
  LongRec(FBIOSCounter).Lo :=
    MemW[BIOSArea:BIOSCounterLo];
  LongRec(FBIOSCounter).Hi :=
    MemW[BIOSArea:BIOSCounterHi];
  Result := FBIOSCounter;
end;
{$endif}

{$ifdef VERSION3}
function TBIOSCounterForm.GetBIOSCounter: Longint;
begin
  WordRec(LongRec(FBIOSCounter).Lo).Lo :=
    Mem[BIOSArea:BIOSCounterLo];
  WordRec(LongRec(FBIOSCounter).Lo).Hi :=
    Mem[BIOSArea:Succ(BIOSCounterLo)];
  WordRec(LongRec(FBIOSCounter).Hi).Lo :=
    Mem[BIOSArea:BIOSCounterHi];
  WordRec(LongRec(FBIOSCounter).Hi).Hi :=
    Mem[BIOSArea:Succ(BIOSCounterHi)];
  Result := FBIOSCounter;
end;
{$endif}

{$ifdef VERSION4}
function TBIOSCounterForm.GetBIOSCounter:
  Longint; assembler;
asm
  mov ax, BIOSArea
  mov es, ax
  mov di, BIOSCounterLo
{ Longint's are returned in DX:AX }
{ AX is the low result word }
  mov ax, es:[di]
  mov di, BIOSCounterHi
{ DX is the high result word }
  mov dx, es:[di]
  les di, Self
{ Having loaded the reference to this form in ES:DI, we
  need to access its FBIOSCounter field. We then need to
  store AX in its low word. After that we get the high
  BIOS timer counter word and do the same thing but
  store AX in the high word. An example of a statement
  that does this is:
  mov TBIOSCounterForm(es:[di]).FBIOSCounter.Word[0], ax
  We can consider this split into two parts. The part
  that references the field of interest, i.e. structured
  variable access, and the part that accesses the
  relevant word of the field, i.e. unstructured variable
  access. The structured variable access can be written
  in these ways:
       TBIOSCounterForm(es:[di]).FBIOSCounter
       TBIOSCounterForm[es:di].FBIOSCounter
       TBIOSCounterForm([es:di]).FBIOSCounter
       (TBIOSCounterForm ptr es:[di]).FBIOSCounter
       (TBIOSCounterForm ptr [es:di]).FBIOSCounter
       ([TBIOSCounterForm ptr es:di]).FBIOSCounter
       es:TBIOSCounterForm[di].FBIOSCounter
       es:TBIOSCounterForm([di]).FBIOSCounter
       es:[di].TBIOSCounterForm.FBIOSCounter
       [es:di].TBIOSCounterForm.FBIOSCounter
  The unstructured variable access can be written in
  many different ways (the *'d formats are valid if you
  are accessing only the first bytes):
       LongRec(structured_part).Lo
    *  Word(structured_part)
    *  structured_part.Word
       structured_part.Word.0
       structured_part.Word[0]
    *  word ptr structured_part
    *  word ptr [structured_part]
       word ptr structured_part + 0
       word ptr [structured_part] + 0
       word ptr [structured_part + 0]
  So, to get to the low and high word of this field the
  following two lines are valid possibilities:          }
  mov word ptr [(TBIOSCounterForm ptr
    es:[di]).FBIOSCounter] + 0, ax
  mov es:[di].TBIOSCounterForm.FBIOSCounter.Word.2, dx
end;
{$endif}

procedure TBIOSCounterForm.FormActivate(Sender: TObject);
begin
  repeat
    Application.ProcessMessages;
    Label1.Caption := IntToStr(BIOSCounter);
  until Application.Terminated;
end;
end.
```

form (FBIOSCounter) before returning the value.

The four GetBIOSCounters have conditional compilation directives around them. None of them will compile until the appropriate symbol for them is defined. You'll notice that just before the first one, I have defined the first symbol with another compiler directive to enable the first version to compile. To get each successive version to compile, just change the {$define VERSION1} statement to define the relevant symbol instead.

There are three issues that we need to understand before we can look at the implementations of the methods. Firstly, where is the BIOS data segment? Secondly, where is it when running in protected mode? And lastly, how do we read a value from memory anyway?

To answer the first and second questions, the BIOS Data Area lives at segment number $40 when running under real mode, in DOS, and the BIOS counter takes up the two words starting at offset $6C and $6E in that segment. In protected mode, we don't have segments, we have selectors instead. A selector has no indication of what real segment it represents and so normally we have to use special interrupt routines or Windows API calls to generate a selector for a particular segment.

However, under Windows 3.x, due to what we can consider to be more than just a happy coincidence, selector number $40 just so happens to represent segment $40. Microsoft programmed this in because of the amount of code under DOS that referred to the BIOS Data Area, in order to act as a safety net for those developers who forgot to rewrite the relevant bits of code. I have seen this referred to as the "save your butt" selector. You should not use this selector; you cannot rely on its value in any particular version of Windows. I wouldn't use it, except for the brevity factor in this code listing. Honestly.

To dispense with the third question, we can read directly from memory using the scarcely documented Mem, MemW and MemL pseudo-arrays. These take a selector and offset as an index and return the Byte, Word or Longint respectively that is stored at that address. The first version of the routine is fairly straightforward. There is no typecast, simply an assignment of the double-word value read using MemL to the data field.

➤ *Figure 2 Assembly in Delphi, smooth as clockwork*

The second one is more interesting. The two individual words are read one at a time, using MemW, and assigned to the relevant word of FBIOSCounter using variable typecasts. The third version is more involved. Each byte is read in turn, using Mem, and assigned to the relevant byte in the data field using nested variable typecasts. We access each word using a LongRec typecast, and then access each byte of the returned word using a WordRec typecast.

The last one is where things start to get messy. It is implemented entirely in assembler code. To achieve what we want in assembler, after reading the values from the BIOS Data Area we need to find a pointer to the current object (ie Self) and load it into appropriate registers, eg ES and DI. Then we typecast this generic register pointer into a pointer to the relevant class type, TBIOSCounterForm, so we can de-reference the FBIOSCounter data field. That's one typecast, and a variable one at that. That gives us access to the double word data field. The trouble is, the standard CPU registers are 16-bits wide and the current Delphi built-in assembler certainly only understands 16-bit assembly operations directly (although you can fool it). So given the structured typecast to give us the data field, we now need to apply a second typecast to get access to the individual words of the data field.

Without going into too much detail, the comments in the listing point out that there are a massive ten (count them) ways to express the first typecast to get us to the data field. And then there are another ten ways to access a specific word. Given that we need one from each set of typecasts to get access to either word in the data field that gives a colossal one hundred combinations of valid typecasts that are understood by the assembler to achieve the end result. I need to sit down after all that....

Brian Long is an independent consultant and trainer specialising in Delphi. His email address is 76004.3437@compuserve.com

A note from the Editor...
For more information on allocating and using selectors, and also on a wide range of subjects including the Pascal expression parser, typecasting, direct memory access etc, see *The Borland Pascal Problem Solver* by Brian Long, published in 1994 by Addison-Wesley, ISBN 0-201-59383-1. Admittedly, it was written with Borland Pascal in mind, but many of the concepts and ideas transfer readily across. I can recommend it.