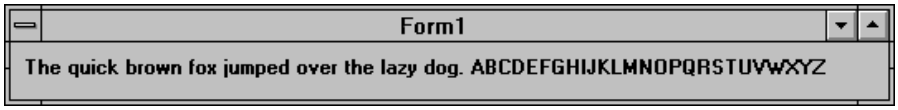
Subclassing Windows

by Brian Long

So what is subclassing? Well, the answer depends on the context of the question. It means different things to different people. Let's look at definitions from some industry gurus.



➤ *Our all-singing, all-dancing example!*

Firstly Bjarne Stroustrup, the guy who designed the C++ programming language, in his book *The C++ Programming Language, Second Edition*, says: "A base class is sometimes called a superclass and a derived class a subclass." He goes on to mention that this is a confusing definition given that "an object of a derived class has its base class as a subobject and also that a derived class is larger than its base class in the sense that it holds more data and provides more functions." From this we learn that in the OOP world, subclassing can be taken to mean deriving new objects.

As for Charles Petzold, in *Programming Windows 3.1* in a paragraph discussing scroll bars says "the window procedure for the scroll bar controls is somewhere inside Windows. However, you can obtain the address of this window procedure by a call to `GetWindowLong` using the `GWL_WNDPROC` identifier as a parameter. Moreover, you can set a new window procedure for the scroll bars by calling `SetWindowLong`. This technique, called 'window subclassing' is very powerful."

So a subclass is a derived object, but window subclassing (or more correctly, window instance subclassing, as there is also a concept of global window subclassing) involves changing the functionality of a window/control. The reason we get (at least) two definitions is that the base term *class* is an OOP term, but it is also the term Microsoft chose to apply to a set of information that is required when creating a window. Most important of this information is the window procedure, the subroutine that responds to messages sent to the window, dictating how the window will appear and function.

With tools such as Delphi, the window procedure is tucked away under the plush carpet of the class library, but Delphi does not prevent us accessing it directly.

Typically when people use the term subclassing loosely, it ends up meaning a combination of the two ideas mentioned above: changing the behaviour of a particular window instance by deriving a new object class. This ends up being the most convenient way of changing the functionality accessed by the window procedure associated with the window class of the window.

So with all that borne in mind and with Delphi both giving us high level encapsulations of all things Windows-based and also allowing us to get to the low-level nuts and bolts of Windows, what options do we have for subclassing a Window? The answer is several.

To explore them all, let's take a trivial task and implement the relevant subclassing in as many different ways as possible. The task will be writing to the caption of a label as characters are typed on the keyboard when the form in a simple application has the focus, in other words changing the default functionality that occurs when the form window receives a `wm_Char` message. However, before we start it will be useful to view the course a message takes as it is digested through a Delphi application.

Message Execution Flow

When Windows has a message that needs to be delivered to an application, it normally places it in the application's message queue (which defaults to a capacity of eight messages). At least this is the case if the message was sent using `PostMessage`. If `SendMessage`, or the Delphi method `Perform`, were used instead, the message is passed directly to the appropriate window procedure by Windows/Delphi.

Messages which are sent using `PostMessage`, or which manage to get into the application message queue in more elaborate ways, are called *queued* messages. Those that go directly to the window procedure are *unqueued* messages.

The mechanism by which queued messages get delivered to the appropriate window in the program is wrapped up in an `Application` object method called `ProcessMessages`, normally called from `Application.HandleMessage` (itself called repetitively from `Application.Run`), but is also called by the yielding method `Application.ProcessMessages` .

Inside `ProcessMessages`, when a message is plucked from the queue, it is given to the `Application` object's user-supplied `OnMessage` handler if one exists, which has the option of terminating the message's existence, if it deems it appropriate. If the message survives this first hurdle, it is sent to the appropriate window procedure using the Windows API function `DispatchMessage`.

The window procedure of the form or control (or any object descended from `TWinControl`) that receives the message is a small stub of code which calls the non-virtual method `MainWndProc`, which in turn passes the message straight onto the virtual method `WndProc`, originally defined in the `TControl` object. Each object that

overrides WndProc in the Visual Component Library adds additional default handling. It is in this default handling that the message may again get swallowed. If it makes it through this point, the message gets passed to the TObject method Dispatch, which invokes the dynamic method dispatching system to allow any specific message handlers (defined using the message keyword) that have been implemented to be called. Pre-written message handlers in the VCL have the task of calling any event handlers that have been written at relevant points.

With that meandering tale finished, it's on with the show.

Method 1: The OnMessage Event

If we do this chronologically, the earliest place we can encroach upon the default message handling scheme (for queued messages) is by defining an event handler for the Application object's OnMessage event. Unfortunately we can't ask the Delphi environment to manufacture an OnMessage handler as the Application object does not have a visual representation; we have to do it manually, as shown in Listing 1 in the form's OnCreate handler, FormCreate.

The event handler, MsgHandler, will be triggered as soon as a message is picked from the application queue inside Application.ProcessMessage, regardless of the target window. So in the handler, we must check that the target window handle of the message matches our form's Handle property and that the message is a wm_Char message and then do what we want to do: determine what key was pressed and add it to the label's caption.

Look in the Delphi help file for more on the OnMessage event.

Method 2: The Windows SDK Approach

If you say "subclass" to an experienced Windows SDK programmer, there is a strong possibility that, in a fashion usually associated with Pavlov's dogs, they will involuntarily say "SetWindowLong". In the world of procedural API programming, window instance subclassing is done using SetWindowLong to replace the current window procedure with another one of our choosing. Because of implementation issues of object methods, the new window procedure needs to be a global function, not a method, although we'll see how to overcome this in the next section.

Changing the window procedure is better than using an OnMessage handler since a window procedure deals with all messages, no matter how they are sent. The OnMessage event only reacts to queued messages and so the window procedure is the earliest place in the scheme of things that is guaranteed to see every message.

The example in Listing 2 replaces the form's window procedure with a global routine called NewWndProc. The parameters passed into the window procedure allow us to identify the target window (fairly redundant in this application since this window procedure is associated with only one window; however you may see fit at some point to write one window procedure to be associated with several windows), the message number and the two additional pieces of information associated with the message.

In the form's creation event, SetWindowLong is called with the parameter gwl_WndProc to specify that we wish to change a window procedure and the form's window handle to indicate which window is to be changed. In addition, the address of our replacement window procedure, typecast into a Longint, is also passed along. The return value from SetWindowLong is the address of the old window procedure: we need to call this from our replacement window procedure so we save this address.

In the new window procedure, we check if the message is wm_Char – if it is we update the label, if it isn't we invoke default functionality by using CallWindowProc to call the original window procedure, passing the same parameters as we were given. Notice that in the form's OnClose event handler, FormClose, we tidy up by setting the window procedure back to the original routine that we saved.

Look in the Windows API help file for more on SetWindowLong.

Method 3: Windows SDK ++

In the section above on the course taken by a message through an application, I mentioned that the window procedure for all forms was a small stub of code which calls the MainWndProc method. The

➤ *Listing 1*

```
unit Subu1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TWndProc = function(HWindow: HWND; Message: Word;
    WParam: Word; LParam: Longint): Longint;
  TForm1 = class(TForm)
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
  private { Private declarations }
  public  { Public declarations }
    procedure MsgHandler(var Msg: TMsg; var Handled: Boolean);
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.MsgHandler(var Msg: TMsg; var Handled: Boolean);
begin
  if (Msg.HWnd = Handle) and (Msg.Message = wm_Char) then
    Label1.Caption := Label1.Caption + Chr(Msg.WParam);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage := MsgHandler;
end;
end.
```

reason MainWndProc can't be set as the window procedure directly is that it is a method. Windows expects to be given a function which takes a particular set of four parameters.

The implementation of methods in the Object Pascal language causes them to have an additional hidden parameter, Self, used to identify the currently executing object instance, which would cause any attempt at parameter passing by Windows to become unsynchronised. That is why we have to supply a global routine. However, we can work around this limitation using the function MakeObjectInstance and its partner FreeObjectInstance.

MakeObjectInstance takes one parameter, the name of a method that you wish to use as a window procedure, which needs to be defined as taking a TMessage parameter, a record holding message information. It returns a pointer to a small block of code which can be given to SetWindowLong and which will invoke your window procedure method. When you are done with this window procedure, be sure to call FreeObjectInstance to deallocate this code block.

Both MakeObjectInstance and FreeObjectInstance were present in Borland Pascal 7, but weren't documented. In Delphi the *Component Writer's Guide* tells us what we can use them for. They can be seen as parallels to the Windows API functions MakeProcInstance and FreeProcInstance which deal with procedure instances: small chunks of code that Windows can safely call, which in turn safely call some exported routine in your code. These functions generate code snippets to safely allow Windows to indirectly call object methods.

In Listing 3, MakeObjectInstance is used to turn NewWndProc, a form method, into the form's window procedure. In the form's OnClose event handler, the window procedure is set back to its original value and our window procedure calling stub, which is returned back from SetWindowLong, is freed with FreeObjectInstance.

```
unit Subu2;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private { Private declarations }
  public  { Public declarations }
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
var OldWndProc: TFarProc;
function NewWndProc(HWindow: HWND;   Message: Word;
  WParam: Word; LParam: Longint): Longint; export;
begin
  Result := 0;
  if Message = wm_Char then
    Form1.Label1.Caption := Form1.Label1.Caption + Chr(WParam)
  else
    Result := CallWindowProc(OldWndProc, HWindow, Message, WParam, LParam);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  OldWndProc := TFarProc(SetWindowLong(Handle, gwl_WndProc,
    LongInt(@NewWndProc)));
end;
procedure TForm1.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  SetWindowLong(Handle, gwl_WndProc, LongInt(@OldWndProc))
end;
end.
```

➤ *Listing 2*

```
unit Subu3;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private { Private declarations }
  public  { Public declarations }
    FOldWndProc: TFarProc;
    procedure NewWndProc(var Message: TMessage);
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.NewWndProc(var Message: TMessage);
begin
  with Message do begin
    Result := 0;
    if Msg = wm_Char then
      Label1.Caption := Label1.Caption + Chr(WParam)
    else
      Result := CallWindowProc(FOldWndProc, Handle, Msg, WParam, LParam);
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  FOldWndProc := TFarProc(SetWindowLong(Handle, gwl_WndProc,
    Longint(MakeObjectInstance(NewWndProc))));
end;
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  FreeObjectInstance(
    Pointer(SetWindowLong(Handle, gwl_WndProc, LongInt(FOldWndProc))));
end;
end.
```

➤ *Listing 3*

Method 4: Virtual Window Procedure Method

Having gone to all the trouble of finding how we can do what is essentially passing a method into SetWindowLong, we will now see that there was no real need to worry about it. As mentioned previously, there is a virtual method that acts as a window procedure method already waiting for us to override it. So we can now dispense with SetWindowLong, MakeObjectInstance and FreeObjectInstance.

The code in Listing 4 is pretty straightforward. We override the virtual WndProc method and inside it we update the label if a wm_Char message is received, otherwise we call upon the default WndProc functionality inherited from the TForm object.

Method 5: Message Handlers

In all the cases so far, we have needed to check the message that comes through to ensure it matches the one we are interested in trapping. There is an elegant construct which allows us to forego these comparisons. The message keyword, when used in conjunction with a method definition and an appropriate message identifier, allows us to write a specific message handler. The handler needs to take either a generic TMessage record as a parameter, or one of the specific message records defined in the Messages unit, or alternatively a user-defined message record if this is not a standard message.

In our case we are changing the behaviour of a wm_Char message so the WMChar method uses a specific TWMChar record type. To complete the declaration of the method in the class definition, the message keyword is followed by the wm_Char identifier.

Although this construct allows us to conveniently write a specific message handling method inside the target window class definition, there is a down side to it. Despite the implementation being mostly in hand-optimised assembler, the dynamic method dispatching scheme used in TObject.Dispatch is slower than the virtual method

```
unit Subu4;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TWndProc = function(HWindow: HWND; Message: Word; WParam: Word;
    LParam: Longint): Longint;
  TForm1 = class(TForm)
    Label1: TLabel;
  private { Private declarations }
  public  { Public declarations }
    procedure WndProc(var Message: TMessage); override;
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.WndProc(var Message: TMessage);
begin
  if Message.Msg = wm_Char then
    Label1.Caption := Label1.Caption + Chr(Message.WParam)
  else
    inherited WndProc(Message);
end;
end.
```

➤ *Listing 4*

```
unit Subu5;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TWndProc = function(HWindow: HWND; Message: Word; WParam: Word;
    LParam: Longint): Longint;
  TForm1 = class(TForm)
    Label1: TLabel;
  private { Private declarations }
  public  { Public declarations }
    procedure WMChar(var Msg: TWMChar); message wm_Char;
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.WMChar(var Msg: TWMChar);
begin
  Label1.Caption := Label1.Caption + Chr(Msg.CharCode);
end;
end.
```

➤ *Listing 5*

```
unit Subu6;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TWndProc = function(HWindow: HWND; Message: Word; WParam: Word;
    LParam: Longint): Longint;
  TForm1 = class(TForm)
    Label1: TLabel;
    procedure FormKeyPress(Sender: TObject; var Key: Char);
  private { Private declarations }
  public  { Public declarations }
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  Label1.Caption := Label1.Caption + Key;
end;
end.
```

➤ *Listing 6*

dispatching scheme, as discussed in Chapter 2 of the *Component Writer's Guide*.

An object's virtual method table has pointers to all virtual methods, be they inherited from ancestor objects or new ones. The dynamic method list on the other hand is much more space conservative (necessary since an object can amass message handlers for many Windows messages) and only holds pointers to new methods introduced in the current object. This means that to find the relevant address to jump to, `Dispatch` may need to search through the dispatch lists of all the object's ancestors. Because of this, if speed is key to your application, you would do better to use one of the previous window procedure or message event approaches.

More information on creating message handlers can be found in Chapter 7 of the *Component Writer's Guide*, or alternatively in the Component Writer's help file by searching for the messages section and the topic "Handling messages."

Method 6: Individual Events

After all that, for this particular scenario there is a much simpler and more familiar approach. A Delphi event handler can be manufactured using the Object Inspector's Events page for the `OnKeyPress` event for the form in the normal way, as shown in Listing 6. Inside the VCL, the `OnKeyPress` event's associated routine (or perhaps, more accurately, I should say the method pointed to by the `OnKeyPress` pointer property since the events listed in the Object Inspector are nothing more than properties for storing method addresses in) is called indirectly from a `wm_Char` message handler method in the `TWinControl` ancestor object of the `TForm`.

Conclusions

So, all this goes to show that with an advanced product like Delphi, which a lot of people treat like a clever 4GL, but which of course is really a very capable 3GL in a most sumptuous wrapping, there are more ways than one to skin the proverbial cat. In this particular example of window instance subclassing, we have six different approaches.

We can supply a routine that gets called for all queued messages targeted to any window in the application, though this will miss any nonqueued messages. There are two ways of writing traditional window procedure replacements, using either a global routine or an object method in conjunction with a Windows API function. Then there is the already present virtual window procedure method and also the elegant message dispatch methods. Last but by no means least is the standard Delphi event model. And that, I think, is plenty for one day.

Note: Brian's examples are on the free disk with this issue, as Delphi projects called SUB1 to SUB6.

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