The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Undocumented Delphi 4

Q Since you seem to have known about various undocumented Delphi things in the past, have you learnt of anything interesting in version 4 that didn't make the manuals, on-line help or README.TXT file?

A You're right in that I happen to be aware of a certain amount of undocumented stuff, by keeping my ear close to the ground (as well as other places). For previous examples of undocumented Delphi features, refer to *The Delphi Clinic* in Issues 6, 13 and 36.

Delphi 4 seems to open the floodgates with respect to undocumented settings. Table 1 shows those I have encountered.

In addition to all these useful things, there are Easter Eggs to be found throughout the product. In Delphi 4's own About box you can hold down the `ALT` key and then (keeping it held down) type in the letters that make up some words: see Table 2.

You may recall that in Delphi 1 you could use `Alt+AND` to get a winking picture of Anders Hejlsberg, the original Delphi author. Now `Alt+CHUCK` plays a short video of Chuck Jazdzewski, the chief R&D guy now that Anders has left (see Figure 1).

In the BDE Administrator for BDE 5, you can double-click the About box icon whilst holding `Ctrl+Shift`. Previous versions of the BDE Administrator required you to click on the About box's client area (ie not the group box).

BDE 5's API help file (BDE32.Hlp) also stores developer references. Look up the topic `Credits` to see the names of the development team members.

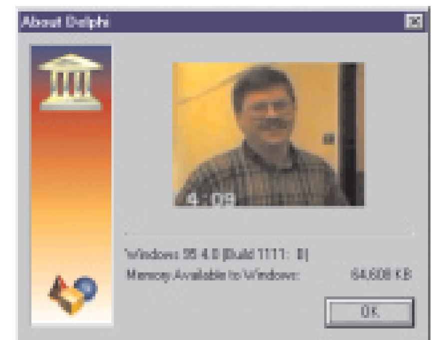
Stopping Windows Shutdown

Q I can't seem to set up an event which would make sure that Windows doesn't close down if my applications are still open. What's direct and simple on this front?

A Well here is an interesting turnaround. In Issue 36 we were looking at how to force Windows to close. Now we want to stop it closing. Anyway, there are two possibilities here.

Firstly, in all versions of Delphi, a form's `OnCloseQuery` event will fire when Windows is closing. This allows you to easily make an event handler to make an appropriate decision: set the `CanClose` parameter to `False` to prevent a closure. However, `OnCloseQuery` will be triggered both if Windows is closed whilst a form is open, and also when a user closes a form explicitly. If you want some code that executes *only* when Windows is being closed, then you will need to write a message handler.

➤ *Figure 1*



➤ *Table 2*

Alt+DEVELOPERS	Gives a scrolling list of the Delphi R&D people.
Alt+QUALITY	Ggives a list of the Delphi QA team.
Alt+TEAM	Gives a list of the developers, QA people, primary US support, documentation writers, translators etc.
Alt+CHUCK	Plays a short video of Chuck Jazdzewski, the chief R&D guy now that Anders has left.

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  CanClose := MessageDlg('Allow form to close?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes
end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
type
  TForm1 = class(TForm)
  ...
    procedure WMQueryEndSession(var Msg: TWMQueryEndSession);
      message wm_QueryEndSession;
  end;
...
procedure TForm1.WMQueryEndSession(var Msg: TWMQueryEndSession);
const
  Prompt = 'Allow program to close, and thereby let Windows session end?';
begin
  LongBool(Msg.Result) :=
    MessageDlg(Prompt, mtConfirmation, [mbYes, mbNo], 0) = mrYes;
  //If it still seems okay to terminate, then call previously installed checking
  if LongBool(Msg.Result) then
    inherited;
end;
```

Command-Line Parameters	
/NS or -NS	No Splash Screen.
/HV or -HV	Heap Verify. When not doing anything better, the IDE calls its `GetHeapStatus` memory management routine. Any heap integrity problems are reported as error codes on the IDE caption bar. These codes are listed in Delphi's SOURCE\RTL\SYS\GETMEM.INC.
/HM or -HM	Heap Monitor. Debug switch that causes Delphi 4's caption bar to display the number of blocks and bytes the IDE has allocated. This makes it easier to find IDE memory leaks.
/attach:<PID>	As the name implies, allows you to run the IDE debugger and attach it to a specified process identifier (PID) under Windows NT. As a follow-on from this, the Delphi 4 IDE can also be used as a Win32 post-mortem, or just-in-time (JIT) debugger. To tell Windows 95 or Windows NT about this, under `\HKEY_LOCAL_MACHINE\ SOFTWARE\Microsoft\WindowsNT\CurrentVersion\AeDebug` set the `Debugger` value to `"C:\Delphi 4.0\ Delphi32.Exe /Attach:%ld"`, using an appropriate path (and yes, that registry path is correct, apparently even if you are using Windows 95). Also, set the `Auto` key to 0 to ensure you are prompted before Delphi 4 is started. Once you have done this, the next time an application crashes, you can press `Cancel` to have Delphi attach to the broken app.
Registry Settings (all these are relative to `HKEY_CURRENT_USER\Software\Borland\Delphi\4.0`):	
Debugging\Enable Attach Menu	Set to a string value of "-1" makes the `Run | Attach to Process...` menu item visible. This generates a dialog full of process ids (PIDs) and you can attach the debugger to any of these processes. This is the same functionality as the `/attach` command line parameter.
Compiling\ShowCodeInsiteErrors	Set to a string value of "-1" causes any Code Insight errors (background compilation errors spotted when trying to invoke the Code Completion window, or Code Parameters tooltip) to be displayed in the editor's message window.
Component Templates\CCLibDir	Set to a string value that is some shareable directory, makes component templates be stored and accessed in that directory, in the file DELPHI32.DCT.
Editor\DefaultHeight	Set to a string value dictates the initial height of the editor window.
Editor\DefaultWidth	Set to a string value dictates the initial width of the editor window.
Editor\Options\NoCtrlAltKeys	Set to a string value of "-1" causes the `View | Debug Windows` submenu items to have their shortcuts removed. All these debug menu items have shortcuts involving `Ctrl+Alt`, eg, `Ctrl+Alt+W` for `View | Debug Windows | Watches`. Many Windows users have desktop shortcuts set up, which will default to also using `Ctrl+Alt` shortcuts. You can therefore easily get ambiguity. For example, you may set up Microsoft Word to launch through `Ctrl+Alt+W`. In Delphi 4, you might press `Ctrl+Alt+W` for the watch window, but you would instead get MS Word popping up onscreen. Removing these shortcuts from the word go, will avoid you getting erroneous applications launched instead of debug windows displayed.
Extras\AutoPaletteSelect	Set to a string value of "-1" causes the component palette to automatically select the page that the mouse is moved over, to save you clicking on the tabs.
Extras\AutoPaletteScroll	Set to a string value of "-1" means that the contents of a component palette page auto-scroll left and right when appropriate. When you have more components on a page of the component palette than can be displayed at any one time, moving the mouse over the left and right scrollers will automatically scroll one way or the other, rather than you having to click on them.
Globals\PrivateDir	Set to a string value that is some shareable directory (eg `C:\Delphi4.0\Private`), makes various Delphi bits and pieces come from this directory. These include menu templates (DELPHI32.DMT), code templates (DELPHI32.DCI), and default project options (DEFPROJ.DOF and DEFPROJ.CFG).
Globals\PropValueColor	Set to a string value (eg `$FF0000`) to make Object Inspector draw property values in this colour.

➤ *Table 1*

To find out if it is okay to terminate, Windows broadcasts a `wm_QueryEndSession` message to all top level windows. You can write a message handler to trap that message, and possibly tell Windows directly that closing is not an option. You can also chain onto the default handling for this message, which is a previously installed `wm_QueryEndSession` message handler in class `TForm`. This VCL message handler is responsible for ensuring the form's `OnCloseQuery` event is set off when Windows is trying to end. Listing 1 shows an `OnCloseQuery` handler (from NoClose.Dpr), and Listing 2 has an implementation of the message handler (from NoClose2.Dpr).

Program Running Upon Windows Start-Up

Q When you shut Windows with Windows Explorer still running, and maybe Microsoft Internet Mail also running, the next time Windows comes up, those applications restart as well. How can I get my programs to do that?

A There are a number of ways to do this, but most of them require more work than is worthwhile. For example, you could add a shortcut to your program to the Startup folder as Windows exits, and make sure you delete it as your program starts up again. But the

best way to do this when running Windows 95 or Windows NT is the way that Windows itself uses. There is a registry key set up for this. What you need to do is to programmatically add a string value for your program under:

```
HKEY_CURRENT_USER\Software\
  Microsoft\Windows\
  CurrentVersion\RunOnce
```

If you write an appropriate value there when Windows is terminating, during its next relaunch Windows will execute all the commands in the `RunOnce` section after logging in, and then delete them. On the other hand, if you add a value into

```
HKEY_LOCAL_MACHINE\Software\
  Microsoft\Windows\
  CurrentVersion\RunOnce,
```

Windows will execute the program before logging in (waiting for the program to finish before doing so).

We saw earlier the use of `wm_QueryEndSession` to potentially stop Windows closing. Another message, `wm_EndSession`, gets sent around to the top level windows in all applications if Windows is actually about to close, having found that no applications objected. A message handler for this message can be used to write your program's last will and testament into the registry before being terminated. Listing 3, from the sample project Restart.Dpr, contains a possible implementation.

IDE Keyboard Problem

Q I have just installed Delphi 3 on a new machine. Whenever I use any arrow keys or the PgUp/PgDn keys I get very strange behaviour: extra Editor windows appear. If I hold down the arrow keys, I get lots of editor windows displayed. It is driving me crazy, so I am trying to investigate. How can I write a program to test that the keys on my keyboard are sending the right information to Windows?

A That sounds very irritating. I won't be offering to swap machines with you ☺. It sounds like maybe the keyboard is generating unexpected characters, or there is some system-wide macro system that is replacing the direction keys with various other key presses.

Try the simple program in Listing 4, then push the various keys on your keyboard. The textual names of the keys will be written onto the form's caption bar.

`GetKeyNameText` is an almost annoyingly convenient API under these circumstances. It takes the `LParam` of a keyboard message (`wm_KeyDown` in this case, where the appropriate field of the message cracker record that I am using is called `KeyData`) and generates an appropriate descriptive piece of text based on the current installed language. This should help clarify what an average program takes these cursor movement keys to be.

Figure 2 shows the inanely simple user interface of the program after pressing my right shift key.

Keystroke Interception

Q Could you explain how to stop my PC from beeping after trapping keys in my form's `OnKeyDown` event? I assume that whatever I trap there still gets trapped in some other event, with the beep indicating 'Invalid Key'. I have cleared the key value in the `OnKeyDown` handler by setting the `Key` parameter to zero but it still happens? Specific keys I am trying to trap include `Ctrl+S` and `Alt+S`.

A These keyboard events seem to be regularly misunderstood. Maybe it's time for a thorough investigation into how they work, both under the hood from the Windows messaging level, and also from the higher Delphi event level.

First of all, I would like to say to any newcomers to Delphi that the `OnEnter` event has nothing to do with the `Enter` key. It does not fire when the `Enter` key is pressed. Instead it is an event that

```
type
  TForm1 = class(TForm)
    ...
  public
    procedure WMEndSession(var Msg: TWMEndSession);
      message wm_EndSession;
  end;
...
uses Registry;
...
procedure TForm1.WMEndSession(var Msg: TWMEndSession);
const
  Restart = 'Software\Microsoft\Windows\CurrentVersion\RunOnce';
begin
  if Msg.EndSession then begin
    with TRegistry.Create do
      try
        //If you want to run your app before any user
        //logs in then uncomment the next line of code
        //RootKey := HKEY_LOCAL_MACHINE;
        if OpenKey(Restart, True) then
          //Write a value with an arbitrary name,
          //But the full path to your exe as a value
          WriteString(Application.Title, Application.ExeName)
      finally
        Free //Destructor calls CloseKey for me
      end;
    Msg.Result := 0
  end;
  inherited
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
type
  TForm1 = class(TForm)
  public
    procedure WMKeyDown(var Msg: TWMKeyDown);
      message wm_KeyDown;
  end;
...
procedure TForm1.WMKeyDown(var Msg: TWMKeyDown);
var KeyName: array[0..255] of Char;
begin
  if GetKeyNameText(Msg.KeyData, KeyName, SizeOf(KeyName)) > 0 then
    Caption := StrPas(KeyName)
  else
    Caption := 'Oops - it''s not working...'
end;
```

➤ Figure 2

represents the associated control acquiring input focus. Controls can gain input focus in many ways, such as being tabbed to, clicked in, or whatever (which of course will trigger many other events). However, when they gain focus, the `OnEnter` event indicates this.

Having got that point clear, the three keyboard-related events in Delphi are `OnKeyDown`, `OnKeyUp` and `OnKeyPress`. These are high level representations of certain Windows messages sent to the control that has keyboard input focus, when keys are pressed on the keyboard.

Let's first of all see if we can gain an understanding as to how all these messages (about half a dozen) operate at a Windows level before concerning ourselves with the event management and how to stop keystrokes.

Scenario 1. Let's take the case of someone looking at a form with one edit control on it. The edit has input focus, and the user presses the letter S on their keyboard (and then quickly releases it). The following sequence describes what happens, message-wise, within this scenario.

1. When the key is pressed a `wm_KeyDown` message is placed in the application's message queue, with information indicating that it is aimed at the edit control.

2. At some point later, maybe before or after the key is released (this is unimportant), probably when the application is not doing anything much, the application message processing loop removes the `wm_KeyDown` message from the message queue. The message processing loop is found in `TApplication.ProcessMessage`, a routine which is called by both `Application.ProcessMessages` and `Application.HandleMessage`.

3. Before doing anything useful with the message, `ProcessMessage` passes it to an API called `TranslateMessage`. The prime job of this API is to take virtual key messages (such as `wm_KeyDown`) and manufacture an additional character message. So now there is another message in the application message queue, targeting the edit: `wm_Char`. This `wm_Char` message will represent the ANSI character 's' (or 'S' if Caps Lock was on).

4. The original message is now passed to `DispatchMessage` and so gets sent to the window procedure inside the edit control for any further processing that may be required.

5. The `wm_Char` message is plucked from the message queue and dispatched to the edit control. As it goes, this (and any other) message is also passed through `TranslateMessage`, which ignores most of them.

6. When the key is released a `wm_KeyUp` message is placed in the application's message queue, with information indicating that it is aimed at the edit control.

7. The `wm_KeyUp` message is plucked from the message queue and dispatched to the edit control.

So, in summary, one single key press generates three messages.

The `wm_KeyUp` and `wm_KeyDown` messages come replete with a *virtual key code* to indicate which key was pressed. Virtual key codes are defined in the 16-bit Delphi `WinTypes` unit and the 32-bit `Windows` unit. For example, `vk_Space` (space bar), `vk_Return` (Enter key), and `vk_Menu` (Alt key). There are constants defined for all the keys on the keyboard except the alphanumeric keys. For those keys, you can get the appropriate value by passing the appropriate character to the `Ord` function. For letter keys, use the uppercase letter. So the virtual key codes for keys 1, 2, a, and b are `Ord('1')`, `Ord('2')`, `Ord('A')` and `Ord('B')` respectively.

The `wm_Char` message is generated for keys that are mapped to ASCII/ANSI characters by the keyboard driver (so it won't occur for the function keys, for example). When it is generated, it comes with a character code for the key that was pressed. So the state of the Caps Lock indicator will affect the character code generated by a single letter key press (you may get an upper or lower case character). This is in contrast to the previously described virtual key code messages, where the virtual key code for any key is always consistent.

I know that all this message stuff might seem reasonably irrelevant at the moment, but bear with me. We will examine a few other scenarios before seeing how the events bubble up from these messages.

Scenario 2. Instead of pressing just `S`, the user presses `Shift+S`, where Caps Lock is off. With regard to physical key activity, this means that `Shift` is pressed, `S` is pressed, `S` is released and finally `Shift` is released. The stream of messages generated here is:

```
wm_KeyDown : vk_Shift
wm_KeyDown : Ord('S')
wm_Char    : 'S' or #83
wm_KeyUp   : Ord('S')
wm_KeyUp   : vk_Shift
```

Scenario 3. The user presses `Ctrl+S`. This produces these messages:

```
wm_KeyDown : vk_Control
wm_KeyDown : Ord('S')
wm_Char    : #19
wm_KeyUp   : Ord('S')
wm_KeyUp   : vk_Control
```

Notice that `wm_Char` comes with character 19 (S is the 19th character in the alphabet) because of the use of the Control key. This is a bit like what happens at the DOS prompt. Pressing Ctrl+M generates ASCII character 13, since M is the 13th character.

Scenario 4. This final example assumes Caps Lock is off and `Alt+S` is pressed:

```
wm_SysKeyDown : vk_Menu
wm_SysKeyDown : Ord('S')
wm_SysChar    : 's' or #115
wm_SysKeyUp   : Ord('S')
[various menu-related messages]
...
wm_KeyUp      : vk_Menu
```

Note that since Alt (vk_Menu) is being used, most of the messages have now changed. wm_KeyDown is now wm_SysKeyDown. wm_KeyUp can now be wm_SysKeyUp. Also wm_Char changes to wm_SysChar (again it is TranslateMessage that generates wm_SysChar messages from appropriate wm_SysKeyDown messages).

When a given control has input focus, an OnKeyDown event will occur if either wm_KeyDown or wm_SysKeyDown are received, whilst an OnKeyUp event will occur if either wm_KeyUp or wm_SysKeyUp are received. An OnKeyPress event will be triggered for a wm_Char message but *not* a wm_SysChar message. This last point becomes important and causes special case handling to be required as we will see later on.

Additional functionality provided by Delphi means that if the control is on a form whose KeyPreview property is set True, the form's OnKeyDown, OnKeyPress and OnKeyUp events will be triggered before those of the actual control. So the form can sneakily preview all the key events of the components on the form and do extra stuff such as treat the Enter key like a Tab key (make the Enter key shift input focus from control to control as the Tab key normally does). For more information on that particular topic, refer to *Intercepting Keystrokes* in *The Delphi Clinic* from Issue 8.

So, back to the problem. The questioner's plan is to write a form-wide OnKeyDown handler and, for certain keystrokes that are deemed important, special functionality will be executed. However, since those keystrokes have been handled by the form, the key event needs to be 'killed off' so the underlying control won't see it. Let's say the keystrokes in question are Escape, Ctrl+S and Alt+S. Note that you need to be careful of keystroke clashes. Ctrl+S might be a shortcut for a menu item. Alt+S might correspond to the shortcut of a button, or a label with an associated focus control, or might normally drop down some menu.

In this example's case, Escape will be used to change the colour of the form, Ctrl+S will minimise the program and Alt+S will write something on the caption bar. Our first stab, matching the outline in the question, is the OnKeyDown handler shown in Listing 5. You will notice that the Shift parameter comes in handy for detecting whether Ctrl, Alt and/or Shift are currently being held down.

The trouble with this code (which can be found in StopKey1.Dpr) is that, in addition to doing what we want, it also produces the irritating beep that is normally associated with invalid keypresses, just as the questioner stated. So what's going on here?

Well, setting the Key parameter to zero in an OnKeyDown handler is designed to prevent the default behaviour executing. This means that if you set Key to zero in the OnKeyDown handler of a form whose KeyPreview is True, the original control will not process the originating wm_KeyDown (or wm_SysKeyDown message) meaning that its OnKeyDown event handler will not execute, and neither will any default message handling. If you set Key to zero in a control's OnKeyDown handler, it will execute your code in the event handler, but then will not do any extra message handling that it would normally have done.

➤ *Listing 5*

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  { Check for Escape }
  if (Key  = vk_Escape) and (Shift = []) then begin
    Color := RGB(Random(256), Random(256), Random(256));
    Key := 0
  end;
  { Check for Ctrl+S or Alt+S }
  if (Key = Ord('S')) and ((Shift = [ssCtrl]) or (Shift = [ssAlt])) then begin
    if Shift = [ssCtrl] then
      Application.Minimize
    else
      Caption := 'Alt+S was pressed at ' + TimeToStr(Time);
    Key := 0
  end
end;
```

Most people assume that this means the keystroke is thrown away. This is not true. Remember that the key down action results in two messages in many cases: the wm_[Sys]KeyDown message and a wm_[Sys]Char message. So you might have stemmed the execution flow of one message, but the other still waits in the wings to take over.

In fact most key responses (like the default beep when unrecognised) come from the wm_Char message handling. So the more correct way to pick up a keystroke and then pretend it didn't occur is to use the OnKeyPress event, where possible. The principal exceptions to this rule are the keys that don't generate a wm_Char message (such as function keys).

There are possible problems to be seen in using an OnKeyPress event handler. If we look at the event types (in Listing 6), we should see what these might be. Firstly, OnKeyPress doesn't take a Shift parameter to indicate which of the standard shift-like keys are down. To overcome this problem we can use GetKeyState to find the state of each key individually. Alternatively we could use GetKeyboardState to find them all out at once. Listing 7 has a utility routine that calls GetKeyboardState and then manufactures a value of type TShiftState, just like the Shift parameter. You can see in the listing that if the high bit of the array element corresponding to a key's virtual key code is set, then the key is down.

Another issue between OnKeyDown and OnKeyPress is that the Key parameter is now a Char, to correspond with the character code that comes along with a wm_Char message. So character case needs to be taken into account (hence the use of the UpCase function), as does parameter type (hence the use of the Chr function).

StopKey2.Dpr tries to fix the problems in StopKey1.Dpr by moving the existing key handling into the form's OnKeyPress event handler, but to prove a point, F2 is now trapped in the OnKeyDown handler (see Listing 8).

```
TKeyEvent = procedure (Sender: TObject; var Key: Word; Shift: TShiftState)
  of object;
property OnKeyDown: TKeyEvent;
type TKeyPressEvent = procedure (Sender: TObject; var Key: Char) of object;
property OnKeyPress: TKeyPressEvent;
```

➤ *Above: Listing 6*

➤ *Below: Listing 7*

```
function GetShiftState: TShiftState;
var KeyState: TKeyboardState;
begin
  Result := [];
  GetKeyboardState(KeyState);
  if KeyState[vk_Shift]   and $80 <> 0 then Include(Result, ssShift);
  if KeyState[vk_Control] and $80 <> 0 then Include(Result, ssCtrl);
  if KeyState[vk_Menu]    and $80 <> 0 then Include(Result, ssAlt);
  if KeyState[vk_LButton] and $80 <> 0 then Include(Result, ssLeft);
  if KeyState[vk_RButton] and $80 <> 0 then Include(Result, ssRight);
  if KeyState[vk_MButton] and $80 <> 0 then Include(Result, ssMiddle);
end;
```

```
function GetCtrlLetter(Ch: Char): Char;
begin
  Result := Chr(Ord(Ch) - Ord(Pred('A')))  { Take away one less than letter A }
end;
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
begin
  if (Key  = vk_F2) and (Shift = []) then begin
    Caption := 'F2 was pressed at ' + TimeToStr(Time);
    Key := 0
  end;
end;
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
var Shift: TShiftState;
begin
  Shift := GetShiftState;
  { Check Escape }
  if (Key = Chr(vk_Escape)) and (Shift = []) then begin
    Color := RGB(Random(256), Random(256), Random(256));
    Key := #0
  end;
  { When Ctrl+letter is pressed, the character code is the }
  { position in the alphabet held by the uppercase letter }
  if (Key = GetCtrlLetter('S')) and (Shift = [ssCtrl]) then begin
    Application.Minimize;
    Key := #0
  end;
  if (UpCase(Key) = 'S') and (Shift = [ssAlt]) then begin
    Caption := 'Alt+S was pressed at ' + TimeToStr(Time);
    Key := #0
  end
end;
```

➤ *Listing 8*

As was pointed out earlier, when a character message is generated for a Ctrl+letter combination, the character code matches the letter's position in the alphabet (Ctrl+M generates #13, Ctrl+A generates #1). To transform a given letter into this modified value, the simple utility routine `GetCtrlLetter` is used in the program.

Things are getting better now. F2, Ctrl+S and Escape are all trapped successfully by the code, but Alt+S stubbornly insists on beeping. The problem here was hinted at during the exploration of the underlying message model. Even though `OnKeyDown` is triggered for both `wm_KeyDown` and `wm_SysKeyDown` messages and `OnKeyUp` is triggered for both `wm_KeyUp` and `wm_SysKeyUp` messages, `OnKeyPress` is only generated for `wm_Char` and not `wm_SysChar`. Since Alt+S actually generates a `wm_SysChar` message, we are rather stuck now.

Onto the next possibility. To trap Alt+letter combinations you can write new components based on the ones you wish to use, and write a message handler for `wm_SysChar` in each one. But that would be enormously tiresome.

Another possibility would be to write a message handler in the form for `wm_MenuChar`. This is a message sent to the form when Alt+letter is pressed in any control on the form. If you handle this message, the Windows API help suggests that returning a value of 1 in the high word is the most harmless option. StopKey3.Dpr is just the same as StopKey2.Dpr, but the Alt+S handling has moved into a form-based message handler (see Listing 9).

Here's another idea for the Alt+letter shortcuts. Add a button to the form, set its `TabStop` property to `False`, and give it a caption with the appropriate letter preceded by an ampersand. Now move the button off the left hand side of the form so it cannot be seen (set its `Left` property to a large negative value). The button can now be given an `OnClick` handler as required.

This approach has been well used in the days when Inprise were Borland. For example, every copy of Database Desktop has used a hidden button on its About box with a caption of `&I`. This means that Alt+I does something undocumented, which is to show the version number of the version of BDE being used. This was also used in Turbo Pascal for Windows, WinSight and Resource Workshop. Simple idea, huh?

Delphi 4 introduces another possibility with a new form event, `OnShortCut`. This is a new form event that allows you to pick upon keystroke combinations and consider them form-wide events. StopKey4.Dpr uses the `OnShortCut` event handler that can be found in Listing 10. Notice the use of the Forms routine `KeyDataToShiftState` to translate message-supplied keystroke information into a `TShiftState` variable.

And now for a final, generic, option. As if you haven't read enough! Consider, if you will, the Delphi Editor, the Object Inspector, the Watch window, in fact many windows in the IDE. They all react to Alt+F10 to bring up their popup menu. This isn't done by writing `wm_MenuChar` message handlers. Instead they use hidden menu items. If you want to react to a keystroke globally across a form, add a menu item to your form's main menu, give it the shortcut you want to react to, then set its `Visible` property to `False`. Even if you don't want a menu at all on your form, just use one to house these hidden convenience items. Alternatively, add lots of items to a dummy menu, set the dummy menu to be invisible and then forget about the individual items.

StopKey5.Dpr does it in this manner. Each menu item has an appropriate simple `OnClick` handler. The code should be pretty obvious. If it isn't, refer to the sample project on this month's disk.

Now let's summarise. So `OnKeyDown` can successfully process and swallow keystrokes that don't have a corresponding ASCII/ANSI representation. `OnKeyPress` can deal with keystrokes that have do have a mapping into the ASCII/ANSI character set: this includes both single character key presses and those pressed in conjunction with the Ctrl key. Alt+key combinations can be dealt with by a `wm_MenuChar` message handler, or a hidden button.

In general, though, a consistent way could be to use a hidden menu item with a suitable shortcut.

Anyone not had quite enough of this keyboard message analysis yet? I suspect most readers have left by now! Anyway, if you're bold and brave enough and want to check up on why some keys completely circumvent the standard keyboard events (for example the up and down cursor keys) and also how to deal with this situation, refer back to *Lost Messages* in *The Delphi Clinic* in Issue 20. Basically, the VCL inserts extra keyboard processing (and potential message swallowing) between steps 2 and 3) as listed above.

```
type
  TForm1 = class(TForm)
  public
    procedure WMMenuChar(var Msg: TWMMenuChar); message wm_MenuChar;
  end;
...
procedure TForm1.WMMenuChar(var Msg: TWMMenuChar);
begin
  if UpCase(Msg.User) = 'S' then begin
    Caption := 'Alt+S was pressed at ' + TimeToStr(Time);
    LongRec(Msg.Result).Hi := 1 { I've handled this }
  end else
    inherited { I haven't handled this }
end;
```

➤ Above: Listing 9 ➤ Below: Listing 10

```
procedure TForm1.FormShortCut(var Msg: TWMKey; var Handled: Boolean);
var Shift: TShiftState;
begin
  Shift := KeyDataToShiftState(Msg.KeyData);
  if (Msg.CharCode = vk_F2) and (Shift = []) then begin
    Caption := 'F2 was pressed at ' + TimeToStr(Time);
    Handled := True
  end;
  if (Msg.CharCode = vk_Escape) and (Shift = []) then begin
    Color := RGB(Random(256), Random(256), Random(256));
    Handled := True
  end;
  if (UpCase(Chr(Msg.CharCode)) = 'S') and (Shift = [ssCtrl]) then begin
    Application.Minimize;
    Handled := True
  end;
  if (UpCase(Chr(Msg.CharCode)) = 'S') and (Shift = [ssAlt]) then begin
    Caption := 'Alt+S was pressed at ' + TimeToStr(Time);
    Handled := True
  end
end;
```

```
{ Halt any further form updates }
LockWindowUpdate(SomeForm.Handle)
...
{ Various update operations that won't be drawn just yet }
...
{ Cause window to redraw, albeit with a certain amount of flicker }
LockWindowUpdate(0);
```

➤ Above: Listing 11 ➤ Below: Listing 12

```
{ Halt any further form updates }
SomeForm.Perform(wm_SetRedraw, 0, 0)
...
{ Various update operations that won't be drawn just yet }
...
{ Cause window to redraw }
SomeForm.Perform(wm_SetRedraw, 1, 0)
{ Could just use SomeForm.Refresh, but this might be better }
RedrawWindow(SomeForm.Handle, nil, 0, rdw_Frame + rdw_Invalidate +
  rdw_AllChildren + rdw_NoInternalPaint)
```

Dynamic Fonts Update

Q In Issue 18 there was an article by Stewart McSporran about dynamic fonts and you also mentioned dynamically loaded fonts in *The Delphi Clinic* in Issue 35. I have tried using the prescribed techniques of `CreateScalableFontResource` and `AddFontResource` unsuccessfully in Delphi 1. I used the sample TrueType font supplied by Stewart, inserted a call to `AddFontResource('Acce.ttf')` and then broadcast a `wm_FontChange` message. Then I used `Memo1.Font.Name := 'Acce'` in order to select the font into my memo, but it didn't work. Can you enlighten me?

A The key point here is that the font name is not the same as the font's filename. You can find its name by properly installing the font on your machine temporarily (from Control Panel's Fonts option) and seeing what comes up. Stewart's ACCE.TTF font file contains the Accent SF font. So your code statement needs to be changed to `Memo1.Font.Name := 'Accent SF'`.

Erratic MDI Menu Update

In Issue 35, when discussing problems with menus in MDI applications, I used the API `LockWindowUpdate`. From various sources I have now come to the conclusion that this API is basically very unpopular due to the excessive flickering it causes. It seems the general consensus is to use the `wm_SetRedraw` message instead. Thanks go to Ken Strong on CIX for bringing this to my attention. After looking further into this it seems Inprise R & D's own Danny Thorpe also endorses this as the way to go and in fact *The Delphi Magazine* has already presented this information in *Mike's Corner* in Issue 30.

So, some code that looks like Listing 11 can be changed to Listing 12 for better performance.