

# A Key's Odyssey


Peter Below

*This article follows the path of a keystroke message through the VCL. You will learn how the key processing is implemented, how the OnKey events work and what intervention points for the programmer can be found in the whole process. In addition, things like message processing are explained, and you will learn how to trace messages in the debugger from the message loop to their eventual destination.*

1. Introduction.....	2
2. A quick recap on user input handling in Windows.....	2
3. The test application .....	4
4. The VCL message loop .....	5
4.1 First port of call: Application.OnMessage.....	6
4.2 IsPreProcessMessage and the PreProcessMessage method .....	7
4.3 IsHintMsg .....	7
4.4 IsMDIMsg .....	8
4.5 IsKeyMsg .....	8
4.6 IsDlgMsg .....	9
4.7 TranslateMessage and DispatchMessage .....	10
5. Default processing for key messages in the VCL .....	10
5.1 Tracing message flow in the debugger. ....	10
5.2 Handling CN_KEYDOWN.....	12
5.2.1 TWinControl.CNKeyDown .....	12
5.2.2 Shortcut handling in the VCL: Into TWinControl.IsMenuKey .....	13
5.2.3 Parental guidance: the CM_CHILDKEY message .....	15
5.2.4 Navigation, default and cancel buttons: the CM_DIALOGKEY message .....	16
5.2.5 Back to the message loop.....	17
5.3 Handling WM_KEYDOWN.....	17
5.3.1 Key preview on the form level.....	17
5.4 Handling CN_SYSKEYDOWN and WM_SYSKEYDOWN .....	18
5.5 Handling CN_CHAR and WM_CHAR.....	19
5.5.1 Accelerator handling: the CM_DIALOGCHAR message .....	19
5.5.2 Character preview: the WM_CHAR message .....	20
5.6 Handling CN_SYSCHAR and WM_SYSCHAR.....	20
5.7 Handling CN_KEYUP and WM_KEYUP .....	20
5.8 Handling CN_SYSKEYUP and WM_SYSKEYUP .....	21
6. Summary .....	21
Appendix 1 .....	24
Appendix 2.....	27

# 1. Introduction

Have you ever wondered what happens in your Delphi program when the user presses a key? How the OnKey\* events work? How actions and menus manage to catch shortcut keys independently of the currently active control? How default and cancel buttons react to Enter and ESC keys?

Come with me on a voyage through the depths of the VCL, and your questions will be answered. The path will take us through many twists and turns; it jumps over several API crevices and past a number of useful intervention points for control writers and application programmers (these are marked with  in the text). On the way, you'll learn about the workings of message handler methods, and collect some tips on how to navigate through low level assembler code sections in the VCL with the debugger.

This article describes the key message processing as implemented in Delphi 2007 for Win32 VCL forms applications. A few things have changed in this area compared with Delphi 7, but these are mostly additions that I will highlight when we get to them. Most of this code has survived basically unchanged since the times of Delphi 1, a tribute to the robustness of the design. If you are in a hurry or not interested in all the details you can refer to the [outline in the summary](#) for a condensed overview.

But let's start with a bit of background information about the basic mechanism of user input handling in Windows. I'll only deal with windowed (GUI) applications here; console applications work a bit differently.

## 2. A quick recap on user input handling in Windows

In the bad old days of DOS programming a program had to either continuously poll the OS or BIOS for key input, or install hardware interrupt handlers. Handling the mouse or other graphical input devices like pens or tablets was equally complex. Windows changed this picture drastically. Since it is able to run several applications in parallel, the polling model no longer works; the applications would get into a fight over who owns the keyboard, and chaos would reign.

Input from the keyboard or pointing devices, like a mouse, is received first by low-level system drivers (which are interrupt-driven) that turn it into input events, which are added to a system event queue. A program can use certain API functions (`mouse_event`, `key_event`, `SendInput`) to manufacture such (fake, in this case) input events in code. Note that there are some key combinations that already get handled at the level of the keyboard driver, like Ctrl-Alt-Del; those never make it to the system event queue. The OS then tries to figure out which application to hand these input events to. The picture is complicated by things like Windows hooks and system-wide hotkeys, but I'll ignore that here for simplicity's sake.

For keyboard input there has to be a foreground application with an active window. If a control on this window has the input focus it will receive the key event, otherwise the active window will get it. If there is no foreground application the key event is discarded. To pass the event, a key message is put into the foreground application's message queue. To be more precise: it is the message queue of the thread that owns the target window, since every thread can have its own message queue. In a typical Delphi VCL forms application all visual controls and forms are created in the application's main thread, so the key messages will end up in that thread's message queue. From there the thread is supposed to fetch them at its leisure and hand them to the target window for processing. For this purpose the application's main thread needs to have a message loop.

In a classical Windows application this message loop was part of the code a programmer had to write. It usually took the form shown in [Listing 1](#).

```
while GetMessage(Msg, 0, 0, 0) do begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
```

```
end;
```

**Listing 1:** Simple message loop

`Msg` stands for a record of type `MSG` (Delphi calls this record `TMsg`), which holds the handle of the intended receiver window, the message code, and the message parameters, plus some other data that is rarely used. The `DispatchMessage` call passes the message to the receiver window's message handler function ("window proc", in API speak); another piece of code the programmer had to supply for his own windows. This function usually contained a monstrous case statement for all the message codes the window needed to handle, far more than simple keyboard and mouse messages. Fortunately Windows contains default window procs for all built-in control types, and a generic window proc (`DefWindowProc`) that can be called to get standard processing for messages. A lot of messages flow through a window's window proc and only a small part of those (mostly keyboard, mouse, timer and paint messages) will come from the message loop. Most messages are sent directly to the window proc, either by Windows or by the program, since all interaction with a window or control is done eventually through messages.

To make the life of the programmer easier, the OS already classifies the key events into those useful for text input, for menu and shortcut handling, and others. Instead of one generic key event message there are nine:

Message	Description
<code>WM_KEYDOWN</code>	This message is sent to the control with the focus when a key is pressed, unless it is the Alt key or the Alt key is already held down. The key is identified by a virtual key code in the first message parameter ( <code>wparam</code> ). Due to the key auto repeat feature, several <code>WM_KEYDOWN</code> messages may be received before a key up message comes in. The <code>TranslateMessage</code> function called by the message loop will put a <code>WM_CHAR</code> or <code>WM_DEADCHAR</code> message into the message queue if the key generates a character.
<code>WM_SYSKEYDOWN</code>	This message is sent to the control with focus when the Alt key is pressed or if the Alt key is already held down when another key is pressed. If no control has the focus the active window will receive any key press as <code>WM_SYSKEYDOWN</code> , even if Alt is not held down. The key is identified by a virtual key code in the first message parameter ( <code>wparam</code> ). Due to the key auto repeat feature, several <code>WM_SYSKEYDOWN</code> messages may be received before a key up message comes in. The <code>TranslateMessage</code> function called by the message loop will put a <code>WM_SYSCHAR</code> or <code>WM_SYSDEADCHAR</code> message into the message queue if the key generates a character.
<code>WM_KEYUP</code>	This message is sent to the control with focus when a key is released, unless it is the Alt key or the Alt key is already held down. The key is identified by a virtual key code in the first message parameter ( <code>wparam</code> ).
<code>WM_SYSKEYUP</code>	This message is sent to the control with focus when the Alt key is released, or, if the Alt key is held down, when another key is released. If no control has the focus the active window will receive any key release as <code>WM_SYSKEYUP</code> , even if Alt is not held down. The key is identified by a virtual key code in the first message parameter ( <code>wparam</code> ).
<code>WM_CHAR</code>	This message is sent to the control with focus when a key that creates a character is pressed, unless the Alt key is held down. The message is created by the <code>TranslateMessage</code> function called as part of the message loop. The function takes into account not only the virtual key code of the pressed key, but also the state of the Shift and Ctrl modifier keys and any pending dead

Message	Description
	char message. The character is identified by a character code in the first message parameter (wparam). Due to the key auto repeat feature several WM_CHAR messages may be received before a key up message comes in.
WM_SYSCHAR	This message is sent to the control with focus when a key that creates a character is pressed, and the Alt key is held down. If no control has the focus the active window will receive any character entered as WM_SYSCHAR, even if Alt is not held down. The message is created by the TranslateMessage function called as part of the message loop. The function takes into account not only the virtual key code of the pressed key, but also the state of the Shift and Ctrl modifier keys. The character is identified by a character code in the first message parameter (wparam). Due to the key auto repeat feature several WM_SYSCHAR messages may be received before a key up message comes in.
WM_DEADCHAR	This message is sent to the control with focus when a key is pressed that creates a dead character, unless the Alt key is held down. The message is created by the TranslateMessage function called as part of the message loop. The dead character is identified by a character code in the first message parameter (wparam). A dead character is an accent that will be combined with the next character entered to form an accented character, if this is possible. If this is not possible the WM_CHAR message for the next character entered will be preceded by a WM_CHAR message for the accent character itself.
WM_SYSDEADCHAR	This message is sent to the control with focus when a key is pressed that creates a dead character, and the Alt key is held down. If no control has the focus the active window will receive any dead character entered as WM_SYSDEADCHAR, even if Alt is not held down. The message is created by the TranslateMessage function called as part of the message loop. The message behaves like WM_DEADCHAR, with one exception: if the Alt key is released before the second character is pressed the system always generates two WM_CHAR messages, one for the accent, one for the unaccented version of the second character.
WM_APPCOMMAND	This one is not quite like the other key messages. It is not posted to the message loop like other input messages. Instead, it's created by the system's default handling of certain special keys or extra mouse buttons (DefWindowProc is the responsible party), and sent directly to the target window. Microsoft introduced it as a standard way to handle the many new keys now available on multimedia keyboards and also some types of mice. These keys can be used to modify the sound output, navigate in a browser, call up the standard mail applications and more. If not handled by the receiver window the message will get sent to the window's parent until it reaches a top-level window, then to that window's owner, if it has one.

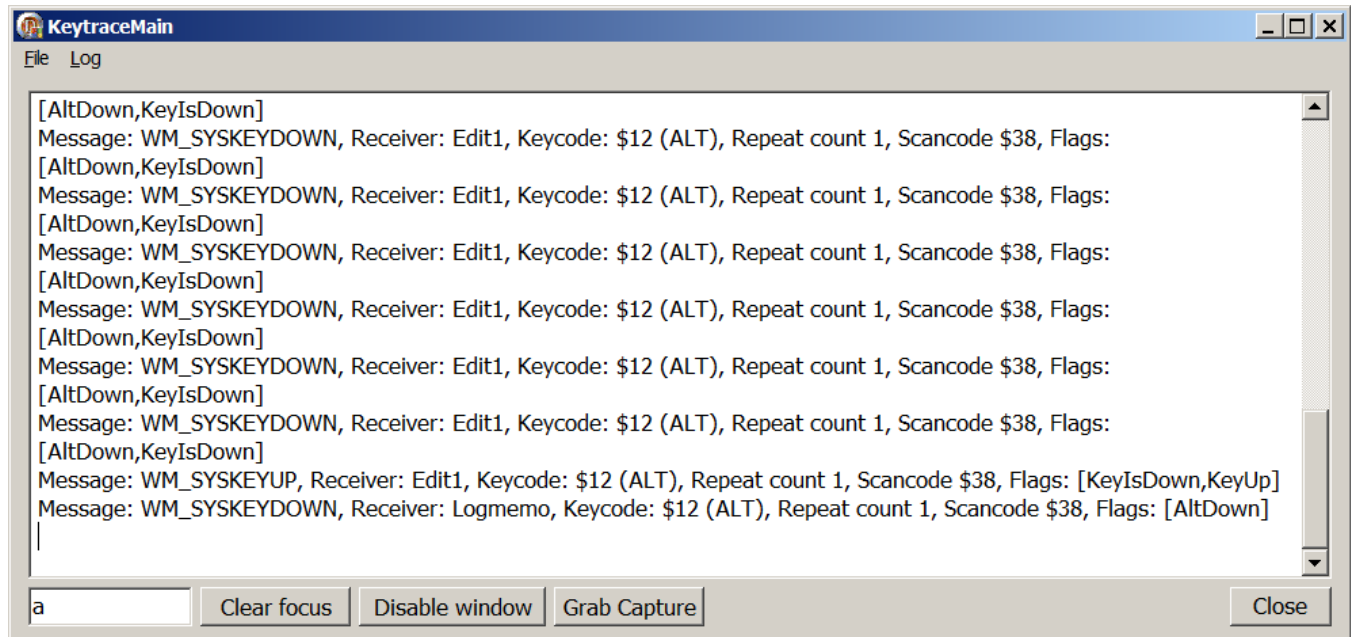
**Table 1:** Keyboard messages

To make this information a bit more concrete, let's build a little test application that can be used to examine the flow of keyboard messages.

### 3. The test application

The test application is fairly simple: a standard VCL form-based application with one form, which contains a large TMemo to display information, a TEdit to take the focus, a close button (TButton)

and some TSpeedbuttons, one of which will remove the focus from the active control so no control in the app has focus any more, and one that will disable the form for 30 seconds, so we can examine how the enabled state influences the key processing. You can download the test application from [CodeCentral](#). It was developed using Delphi 2007.



If none of the log options in the Log menu is activated, the application will just display the key messages with their parameters as they come in. You can use it to get a feeling for the message flow and how it is influenced by having a control with focus or not, or by the enabled state of the form. It is a bit surprising, for instance, to see that a disabled form will still get key messages if it is the foreground window.

You may be confused to see that the repeat count displayed for key down messages is always 1 if you hold down the key to trigger the auto repeat. The help is a bit ambiguous here: the repeat count does **not** count the `WM_KEYDOWN` messages created by the auto repeat. It only will be larger than 1 if the keyboard produces key down events faster than the application processes them. In this case Windows will not add a new `WM_KEYDOWN` message for a key on auto repeat to the queue but increment the repeat count for the last existing message in the queue. On today's machines you will probably never see a `WM_KEYDOWN` message with a repeat count > 1, unless the main thread is busy doing some processing for more than the key auto repeat time.

The application's Log menu allows you to enable or disable additional log features. Many of the intervention points identified in the following text are demonstrated by the application.

## 4. The VCL message loop

As was explained in the second chapter, a Windows GUI application needs a message loop. The VCL provides that for us inside the Application object; the loop is started by calling `Application.Run`, and it will run until the application is terminated. Let's take a look at this message loop; you can find it in the Forms unit in the `TApplication.Run` method:

```
repeat
  try
    HandleMessage;
  except
    HandleException(Self);
  end;
until Terminated;
```

**Listing 2:** VCL message loop

This (pretty self-explanatory) construct makes sure any exceptions raised by the code handling a message are properly trapped and dealt with, by handing them to the `Application.OnException` events handler, if there is one, or by displaying the standard exception message dialog.

`HandleMessage` is also very simple:

```
procedure TApplication.HandleMessage;
var
  Msg: TMsg;
begin
  if not ProcessMessage(Msg) then Idle(Msg);
end;
```

`ProcessMessage` will try to get a message from the message queue and process it. If there is no message in the queue it returns false and the idle processing is executed. That is a major chapter in itself. It takes care of things like hint processing, action updating, thread `Synchronize` calls, etc., but that is out of scope for this article. I want to mention only one thing about `Idle`: the last thing it does is call the Windows `WaitMessage` function, which will block the main thread until the next message arrives in the message queue. Also note that Delphi 2006 finally offered a way to trigger the idle-processing from code: there is a public `DoApplicationIdle` method in `TApplication` for this purpose. But enough of these digressions, let's look at `TApplication.ProcessMessage` next.

This method has seen some revision since Delphi 7, it now checks first whether the target for a message is a Unicode window (registered with `RegisterClassW` or `RegisterClassExW`) or not, and then uses either the widestring versions of `PeekMessage` and `DispatchMessage` or the ANSI versions. This allows Unicode-enabled 3rd-party controls to work properly, even though the VCL itself (for Win32 applications) still uses ANSI windows (registered via `RegisterClassA`) only. A single glance at the `ProcessMessage` method also reveals that it does a lot more than the minimal message loop given in [Listing 1](#). The salient part is quoted in [Listing 3](#), reformatted a bit for clarity, and fitted with line numbers for reference. Note that the VCL is supposed to be compiled with complete boolean evaluation off, so the evaluation of the if conditions will only be done until one of the called functions returns true, making the whole expression false.

```
01:  if Assigned(FOnMessage) then
02:    FOnMessage(Msg, Handled);
03:  if not IsPreProcessMessage(Msg)
04:    and not IsHintMsg(Msg)
05:    and not Handled
06:    and not IsMDIMsg(Msg)
07:    and not IsKeyMsg(Msg)
08:    and not IsDlgMsg(Msg)
09:  then begin
10:    TranslateMessage(Msg);
11:    if Unicode then
12:      DispatchMessageW(Msg)
13:    else
14:      DispatchMessage(Msg);
15:  end;
```

**Listing 3:** Message pre-processing in `TApplication.ProcessMessage`

## 4.1 First port of call: `Application.OnMessage`

The very first step in the VCL's preprocessing of messages is to check whether a handler has been attached to the `Application.OnMessage` event. If this is the case the message is handed to this handler method (lines 1 and 2 in [Listing 3](#)). The test application uses this mechanism to display the key messages as they come in. Keep in mind, however, that this event will see a lot more than only key messages. It sees **all** messages that are posted to the main thread's message queue, which includes all mouse messages, and also timer and paint messages.



The `Application.OnMessage` event is the first opportunity for an application (or its programmer) to intercept a key message. It is not tied to a specific form — the event will see all key messages for all windows created in the main thread — so is the ideal place to implement some central key handling for the whole application. It is even possible to change the message parameters to turn one key into another here. By setting the `Handled` parameter of the event handler to true most (but not all) default processing for a message can be prevented. The message will not reach the target window in this case. However, looking at [Listing 3](#), lines 3 and 4, it is clear that `IsPreProcessMessage` and `IsHintMsg` will still be called.

Setting `Handled` to true for any of the key down messages (`WM_KEYDOWN`, `WM_SYSKEYDOWN`) will prevent the matching character message (`WM_CHAR`, `WM_SYSCHAR`) from being created, since that is done by the `TranslateMessage` API function. However, the key up message (`WM_KEYUP`, `WM_SYSKEYUP`) will still come in, since it is created by the keyboard driver.

## 4.2 IsPreProcessMessage and the PreProcessMessage method

This call was introduced into the VCL message loop (line 3 in [Listing 3](#)) after Delphi 7. Its implementation (check the Forms unit source for details) will call the target control's `PreProcessMessage` method, **unless** some control has grabbed the mouse capture, in which case it does nothing. If no control has the capture, the method tries to determine the VCL control bound to the message's target window handle. If this fails it will walk up the API parent chain until it finds a VCL control or runs out of parents.

`PreProcessMessage` is implemented on the level of `TWinControl` and it is dynamic, so can be overridden in descendent controls. The implementation in `TWinControl` is exceedingly simple:

```
function TWinControl.PreProcessMessage(var Msg: TMsg): Boolean;
begin
    Result := False; { Not handled }
end;
```



The method does nothing and is just there for control writers to override. You can think of it as an equivalent to `Application.OnMessage` on the control level: it will see all messages that go through the message loop and are targeted at the control. For a control user the method is not directly accessible since it is not tied to an event the user could hook. No control in the standard VCL actually overrides this method, by the way. It is in fact a bit redundant since the same messages will also be seen by the control's `WindowProc`. The main difference is that `PreProcessMessage` allows all further processing of the message to be prevented (by returning true).

To see `PreProcessMessage` at work enable the appropriate item in the Log menu, then type a few characters, and then click the "Grab capture" button to set the mouse capture to the form without moving the focus from the edit control. Type again and watch the log: the `PreProcessMessage` method is no longer called.

## 4.3 IsHintMsg

The `TApplication.IsHintMsg` method (called in line 4 of [Listing 3](#)) is the VCL's mechanism to hide any showing hint window when the user performs an input action or the application receives or loses the foreground status. The method is also very simple:

```
function TApplication.IsHintMsg(var Msg: TMsg): Boolean;
begin
    Result := False;
    if (FHintWindow <> nil) and FHintWindow.IsHintMsg(Msg) then
```



```
CancelHint;  
end;
```

Note that the method always returns false, so it will never prevent further processing of a message, even if it considers the message to be a hint message. The test application derives a new class from `THintWindow` and makes it the standard hint window class, to be able to log calls to `IsHintMsg`. Enable the appropriate item in the Log menu to activate this log. A lot of messages go through `IsHintMsg`, so the test app filters out successive calls with mouse move messages and the undocumented \$0118 message, which comes in quite frequently.

`IsHintMsg` does not offer any ready intervention point (other than to replace the default hint window class like the test application does), so is not directly useful for key processing.

## 4.4 IsMDIMsg

This call (line 6 of [Listing 3](#)) is only relevant for MDI (multi-document interface) applications. The Windows MDI framework defines a number of standard key actions for MDI child windows. The ones coming to mind are:

### **Alt+-**

Opens the child windows system menu.

### **Ctrl+F4**

Closes the active child window.

### **Ctrl+F6**

Activates the next child window.

### **Ctrl+Tab**

Activates the next child window.

### **Ctrl+Shift+Tab**

Activates the previous child window.

If the application's main form is a MDI main form and the active form is a MDI child form `IsMDIMsg` will call the API function `TranslateMDISysAccel` for the message, which takes care of these standard shortcuts. A Delphi programmer can just forget about this special treatment. There is no way to hook into this part of the processing.

## 4.5 IsKeyMsg

This call (line 7 of [Listing 3](#)) starts the bulk of the key preprocessing done by the VCL. The method itself (see `TApplication.IsKeyMsg` in `forms.pas`) is deceptively simple; the important statement is this one:

```
if SendMessage(Wnd, CN_BASE + Message, WParam, LParam) <> 0 then  
  Result := True;
```

In fact, the actual code uses `SendMessageA` or `SendMessageW`, depending on whether the target window (`Wnd`) is a Unicode or ANSI window. The target window used may not be the original window passed to `IsKeyMsg` as part of the message record. If another window in the process has the mouse capture it will be used as target for the message, for example. If the original target is not a VCL control the method walks up the API parent chain to find a VCL control and uses it as target. How does the VCL figure out which VCL control a window handle belongs to, by the way? If you want to know the grimy details refer to [Appendix 1](#).

You may have noted that `IsKeyMsg` does not pass on the original key message; it adds an offset (`CN_BASE`) to the message ID to turn the original Windows message into a VCL message.

`WM_KEYDOWN` becomes `CN_KEYDOWN`, `WM_CHAR` becomes `CN_CHAR`, and so on. The VCL message constants are defined in the Controls unit and they all have message IDs above \$B000 (45056 decimal). The naming convention follows the one Microsoft uses in the Windows API: command



messages (messages that instruct the target control to do something) start with `CM_`, notification messages (messages that inform the target control of something it may find interesting) start with `CN_`. If you look at `Controls.pas` you may notice that the list of `CN_` notifications does not contain a notification for every key message in [Table 1](#). Only the notifications actually used inside the VCL are defined.

What do the standard VCL controls do in response to these `CN_` notifications? As it turns out, they do quite a lot and also provide a number of intervention points for the programmer. We will dive into the details in [chapter 5](#). For the moment, keep one important point in mind: if the control returns a value other than 0 as the message result the VCL will consider the key handled and the control will never receive the original Windows message that triggered the notification. All the processing after line 7 in [Listing 3](#) will be skipped. This also means that a `CN_KEYDOWN` notification for a key that produces a character can prevent the creation of a `WM_CHAR` message for the character, since `TranslateMessage` will not be called if the `CN_KEYDOWN` message handler returns a value `<> 0`.

## 4.6 IsDlgMsg

This call (line 8 of [Listing 3](#)) is only relevant if a modeless API dialog is the current foreground window. It deals with navigation between the diverse controls on the dialog. There is a standard in Windows for the keys used to navigate between controls: Tab and Shift-Tab move forward or backward in tab order; if the control with focus does not use the arrow keys for its own purpose these can also be used to navigate around the dialog. Also part of this standard are the Alt-`<character>` shortcuts for buttons, checkboxes, radio buttons, and static text controls. For the latter the shortcut puts focus on the next focusable control in tab order after the static text control.

As an application user you probably took this for granted; as a Delphi programmer using the VCL you never had to waste a thought about this either, since the VCL handles it for you. But for the old-style API level programmer this was a concern indeed, since the Windows API does not handle this navigation automatically. The API offers a special function named `IsDialogMessage` (in ANSI and Unicode versions) that has to be called from the message loop if a dialog is active to get the navigation working.

Most dialogs are used in a modal fashion. Modal dialogs have their own internal message loop and the default loop inside the API functions used to show modal dialogs (`MessageBox`, `DialogBox`, common dialogs, et al.) calls `IsDialogMessage` internally. Modeless dialogs are harder to handle, however. The only place to call `IsDialogMessage` from for those is the application's main message loop, so the program has to keep track of the window handle of the active dialog (if there is one). In days of old this could become a major nightmare in large applications, potentially using many modeless windows.

The VCL uses a different strategy to implement navigation on a form (the `CM_DIALOGKEY` message), but it still needs a way to deal with API-level modeless dialogs. It uses this mechanism for the find and replace dialogs from the `Dialogs` unit. These are common dialogs, part of the Windows API, and shown modeless. The mechanism has two parts:

### **The `Application.DialogHandle` property**

A modeless dialog has to assign its window handle to this property when it is activated and also set the property back to 0 when it is deactivated and the property still has its window handle.

### **The `Application.IsDlgMsg` method**

If `Application.DialogHandle` is not 0 this method will call the appropriate version of `IsDialogMessage` with the dialog's window handle.

There is no way to hook into this step of the VCL's message pre-processing and none is needed. The previous steps already offer a lot of intervention points.

## 4.7 TranslateMessage and DispatchMessage

If the message fetched from the message queue makes it through the gauntlet of pre-processing methods detailed above, it will finally be handed to the API functions `TranslateMessage` and `DispatchMessage` (line 10 to 14 of [Listing 3](#)). The former only processes key down messages. It takes the virtual key code, the state of the modifier keys (Shift, Ctrl, Alt) and any pending dead char message, and manufactures a `WM_CHAR` or `WM_SYSCHAR` message, if the key creates a character (see the API functions `ToAscii`, `ToAsciiEx`, `ToUnicode`). This message is put into the message queue. The `DispatchMessage` function finally calls the target window's window function with the passed message. The window function's return value is discarded since there is no way to pass it back to the message sender for a posted message.

## 5. Default processing for key messages in the VCL

As shown in [chapter 4.5](#), the VCL sends key notification messages (`CN_*`) to a control (not necessarily the one having the focus) before the actual key message (`WM_*`) is delivered. Where in the VCL are these messages processed? We can try to figure this out by tracing the key processing in the debugger. But how do we get through a `SendMessage` call? The following chapter gives a rundown of such a debugging session, and how to deal with the assembly routines we will encounter on the way.

### 5.1 Tracing message flow in the debugger.

Load the Keytrace demo project into the IDE, make sure the project options have the "use debug DCUs" checkbox checked on the compiler page and the "build with run-time packages" checkbox unchecked on the packages page. Build the project. Open the Forms unit, find the `TApplication.IsKeyMsg` method, set a breakpoint on the first `SendMessageA` (or `SendMessage`) call you see. For Delphi 2007 this would be on the line marked with `==>` in the following code snippet:

```
if IsWindowUnicode(Wnd) then
begin
  if SendMessageW(Wnd, CN_BASE + Message, WParam, LParam) <> 0 then
    Result := True;
  ==> end else if SendMessageA(Wnd, CN_BASE + Message, WParam, LParam) <> 0
then
  Result := True;
```

Run the application from the IDE (F9), wait for the form to come up, and type a letter key, e.g. "A". That should stop the application on the breakpoint. Now use F7 to step into the `SendMessage` call. This gets you into the Windows unit, to the line

```
function SendMessageA; external user32 name 'SendMessageA';
```

Press F7 again. This gets us to the `StdWndProc` function in the Classes unit. If you want to trace message flow this function will quickly become quite familiar, since it is the common entry point for message dispatching from the API side into the VCL. It is implemented in assembler, but don't panic; you don't need to understand what it does to proceed with your mission. Just hit F7 until you are on the `CALL` statement, then once more to trace into the call. This now gets us into the Controls unit; we end up in `TWinControl.MainWndProc`. At this point you may want to add a watch for "Name" so you can figure out which control is the one getting the message. It should be "edit1" at this point.

```
01: try
02:   try
03:     WindowProc(Message);
04:   finally
```

```

05:      FreeDeviceContexts;
06:      FreeMemoryContexts;
07:  end;
08: except
09:   Application.HandleException(Self);
10: end;

```

**Listing 4:** TWinControl.MainWndProc

As you can see, the method basically just passes the message on to `WindowProc`. If you check the VCL documentation you will see that `WindowProc` is not a method; it is a public property inherited from `TControl`. It stores a method pointer of type `TWndMethod`. We will see in a moment what the default value of this method pointer is, but the important point is that we have another opportunity for intervention here:



The `WindowProc` property can be used for VCL-style subclassing of controls. Subclassing, in API lingo, is a method to tap into a control's message flow by replacing its window function with one written by the programmer. This way one can change a control's behavior without needing access to the control's source code. The VCL's equivalent is replacing the content of a control's `WindowProc` property with a suitable method of another class, often a form. With both styles of subclassing it is of utmost importance to pass any messages not processed by the replacement window function to the original it replaced. Without that the control will cease to work properly.

The sample application subclasses the `edit1` control and will log messages passing through the replacement window proc (`NewEdit1WndProc`) if the corresponding entry in the Log menu has been checked.

Another point worth noting in [listing 4](#) is the fact that the whole method body is wrapped into a `try..except` block which redirects any trapped exception to `Application.HandleException`. This is the VCL's method to make sure that exceptions raised during processing of a message do not propagate up the stack into Windows code. Allowing them to do so could crash the program, so this is an important safety feature, especially if you keep in mind that in a typical VCL forms application practically **all** code is executed in response to messages.

OK, enough of the digressions; let's get on with the tour. Press F7 until you are on the `WindowProc` line and then once more to trace into the call. This first gets us into `NewEdit1WndProc`. Press F7 once more to trace into the call to the old window proc. This gets us to `TWinControl.WndProc`. So this is the default target for the `WindowProc` property. At first glance it looks like a typical API-level window function, with a big case statement. But it lists only a few messages and none of the `CN_*` messages is among them. If you press F7 the execution will jump over the complete case statement and end up on the line

```
inherited WndProc(Message);
```

Before we see where this call will take us there is something to take note of: since `WndProc` is a **virtual** method we have another intervention point here:



Control writers can override the `WndProc` method to implement special handling for messages. This is not directly useful for application programmers, unless you want to override the `WndProc` of a form. Usually it is better to add or replace a message handler method for a specific message of interest, but overriding `WndProc` can be useful if you want to prevent any message processing done by an inherited `WndProc` or a message handler method called from it. We will get into more detail about message handler methods in a moment.

Press F7 again to trace into the inherited `WndProc` call. Not unexpectedly, that takes us to `TControl.WndProc`. This method does some key and mouse handling, but it does not handle the `CN_*` key notifications. Let's use F8 (not F7!) this time to go through the method until the execution reaches the `Dispatch(Message)` call. Press F7 to trace into the call. Horror! Another

assembler method (`TObject.Dispatch`), and one we will revisit during message tracing quite a bit as well.

`TObject.Dispatch` is Delphi's generic message passing routine. Message passing is not limited to `TWinControl` descendents or Windows-style messages — **any** object can receive a message through `Dispatch`. The only requirement is that the data block passed to `Dispatch` (typically a record) has a numeric key (the message ID, a 16 bit integer or word) in the first two bytes. The `GetDynaMethod` call you see in the code for `Dispatch` tries to find a handler for the message in the class's *dynamic method table* (DMT), or in the DMTs of the ancestor classes. If none is found `Dispatch` calls the `DefaultHandler` method (which is virtual in `TObject` and can be overridden by descendants), otherwise it calls the method it found, passing it the data block received by `Dispatch`. If you want to learn more about Delphi's message handling refer to [Appendix 2](#).

Use F8 again to walk through the `Dispatch` method, jumping over the `GetDynaMethod` call, until you end up on one of the `JMP` statements. Press F7 to execute the jump. This finally gets us to the place where the `CN_KEYDOWN` notification message we have been chasing is handled: the `TWinControl.CNKeyDown` method. Let's leave tracing mode for now and examine this method in more detail.

## 5.2 Handling CN\_KEYDOWN

### 5.2.1 TWinControl.CNKeyDown



Since `CNKeyDown` is a message handler method, component writers can replace it in their own `TWinControl` descendents just by redeclaring it. This is a more targeted way to modify the handling of a specific message, compared to overriding `WndProc`. If you replace a message handler method be sure to call the inherited message handler if that is appropriate, before or after you do your own handling of the message.

A revised version of the `TWinControl.CNKeyDown` method can be found in [Listing 5](#). I have removed the confusing `with` statement, wrapped lines for more clarity, and added line numbers and end comments.

```
01: var
02:   Mask: Integer;
03: begin
04:   Message.Result := 1;
05:   UpdateUIState(Message.CharCode);
06:   if IsMenuKey(Message) then
07:     Exit;
08:   if not (csDesigning in ComponentState) then
09:     begin
10:       if Perform(CM_CHILDKEY, Message.CharCode, Integer(Self)) <> 0 then
11:         Exit;
12:       Mask := 0;
13:       case Message.CharCode of
14:         VK_TAB:
15:           Mask := DLGC_WANTTAB;
16:         VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN:
17:           Mask := DLGC_WANTARROWS;
18:         VK_RETURN, VK_EXECUTE, VK_ESCAPE, VK_CANCEL:
19:           Mask := DLGC_WANTALLKEYS;
20:       end; {case}
21:       if (Mask <> 0) then
22:         (Perform(CM_WANTSPECIALKEY, Message.CharCode, 0) = 0) and
23:         ((Perform(WM_GETDLGCODE, 0, 0) and Mask) = 0) and
24:         (GetParentForm(Self).Perform(CM_DIALOGKEY,
```

```

25:         Message.CharCode, Message.KeyData) <> 0)
26:     then
27:         Exit;
28:     end; {if}
29:     Message.Result := 0;
30: end;

```

**Listing 5:** TWinControl.CNKeyDown

In line 4 the method sets the message result to 1 to mark the message as handled. This will be retained if the execution flow leaves the method through one of the many `Exit` statements. If this does not happen the message result is set back to 0 (unhandled) in line 29.

The `UpdateUIState` call in line 5 is a relatively recent addition (first appeared in Delphi 7). You can look at the source for `TWinControl.UpdateUIState` to see what it does. Basically it sends a `WM_UPDATEUISTATE` message to the control's host form if the key pressed is the Alt key or a navigation key (arrow, tab). When Alt goes down a form is supposed to underline the shortcuts in its menu bar, for example. The navigation keys may result in a change of focus, for which the system is prepared by the message (a bit prematurely in my opinion, since the control may process the key itself so the focus is not moved).

Line 6 is a lot more interesting. The `IsMenuKey` method is the mechanism used by the VCL to trigger the shortcut handling for menus and actions. This is a complex subject and deserves its own chapter. There are a few pitfalls in the implementation and also some useful intervention points.

## 5.2.2 Shortcut handling in the VCL: Into TWinControl.IsMenuKey

The body of `TWinControl.IsMenuKey` is shown in [Listing 6](#), again edited a bit to improve clarity and add line numbers.

```

01: var
02:     Control: TWinControl;
03:     Form: TCustomForm;
04:     LocalPopupMenu: TPopupMenu;
05: begin
06:     Result := True;
07:     if not (csDesigning in ComponentState) then
08:     begin
09:         Control := Self;
10:         while Control <> nil do
11:         begin
12:             LocalPopupMenu := Control.GetPopupMenu;
13:             if Assigned(LocalPopupMenu) and
14:                (LocalPopupMenu.WindowHandle <> 0) and
15:                LocalPopupMenu.IsShortCut(Message)
16:             then
17:                 Exit;
18:             Control := Control.Parent;
19:         end; {while}
20:         Form := GetParentForm(Self);
21:         if (Form <> nil) and Form.IsShortCut(Message) then
22:             Exit;
23:         end; {if}
24:         if SendAppMessage(CM_APPKEYDOWN, Message.CharCode, Message.KeyData) <> 0
then
25:             Exit;
26:             Result := False;
27: end;

```

**Listing 6:** TWinControl.IsMenuKey

The first part of the method is fairly straightforward: starting with the current control on line 9, the method walks up the parent chain and looks for a popup menu that is willing to accept the key as a shortcut. If there are no takers the method next asks the control's host form (if it has one; the control may in fact **be** a form) whether it wants to process the key as a shortcut (line 21).

`TCustomForm.IsShortcut` is an interesting method in itself, since it offers us not one but two intervention points:



`IsShortcut` is a virtual method, so we can override it in our form classes to customize the form's shortcut handling. Doing it this way allows us to completely bypass the built-in shortcut handling on the form level, if that is needed, including the form's `OnShortcut` event.



The first thing `IsShortcut` does is to fire the form's `OnShortcut` event. This event is typically used by application programmers to modify the standard shortcut handling of a form. If the event handler's `Handled` parameter is set to true any subsequent processing for the key will be skipped.

If there is no handler for the `OnShortcut` event, or `Handled` returns `False` (the default), the next party asked to handle the key is the form's main menu, if it has one. If the key is still unprocessed after that it gets passed to a local procedure called `DispatchShortcut`. This procedure is used to recursively iterate over the form's components, starting on the form level and drilling down the ownership hierarchy. If the component under investigation is an action list (a `TCustomActionList` descendent) its `IsShortcut` method is called to allow it to examine the key. This is the mechanism used to trigger actions through their shortcuts. Since `DispatchShortcut` recursively enumerates all components it will also find action lists owned by frames or embedded forms, **if** they are owned by the form or another component owned by the form. This may not be the case for an embedded form.



There is a problem here, due to the **sequence** of the enumeration: if a form contains multiple copies of a frame class that contains an actionlist, these actionlists will be visited in creation order, regardless of which frame currently holds the active control. Since all frames use the same shortcuts this may result in the wrong frame processing the shortcut. The same problem applies to embedded forms, of course. If you have such a situation, use the form's `OnShortcut` event or override `IsShortcut` to make sure the frame/form containing the current active control sees the shortcut first. See this newsgroup post for an example: <http://tinyurl.com/62l8rq>. This also applies if you have conflicts between shortcuts used on the host form and the frames or embedded forms.

If nobody expressed an interest in the key, line 24 of [Listing 6](#) sends the key to the Application object for examination in a `CM_APPKEYDOWN` message. The `SendAppMessage` functions just checks `Application.Handle` for 0 before it sends the message via `SendMessage`, as usual. The message is processed in `TApplication.WndProc`, the window function for the Application window. This is a classical window function with a big case statement handling lots of stuff not of immediate interest to our mission. Only two things are noteworthy here. The first is this bit of code right after the begin of the method:

```
for I := 0 to FWindowHooks.Count - 1 do
  if TWindowHook(FWindowHooks[I]^)(Message) then
    Exit;
```



It may not be obvious at first glance, but we have another intervention point here. It is not particularly useful for key processing, but can come in handy if you need to tap into the application window's message processing for some other reason. The `Application.HookMainWindow` method allows you to add a method to the `FWindowHook` list. The method has to have the appropriate signature (function (var Message: TMessage): Boolean) and you have to remove it from the list through `Application.UnhookMainWindow` before the object containing the method dies.



The second part of interest in `TApplication.WndProc` is what it does with the `CM_APPKEYDOWN` message:

```
CM_APPKEYDOWN:
    if IsShortCut(TWMKey(Message)) then Result := 1;
```



It calls `TApplication.IsShortcut`, which in turn first fires the Application object's `OnShortcut` event. This is another opportunity for the application programmer to intervene in the default shortcut processing. Note that the event will only fire for keys that have not been handled as shortcuts in a more local context (the currently active form) already. Otherwise the event works exactly like a form's `OnShortcut` event. The handler method has a `Handled` parameter the code can set to true to prevent further processing of this key.

`TApplication.IsShortcut` does something more, however: if the `OnShortcut` event did not handle the key it is passed to the main form's `IsShortcut` method for examination. This then triggers the whole raft of processing mentioned [a few paragraphs above](#) on the main form



There is a potential problem here: if the currently active form **is** the application's main form this whole processing will be done **twice**! First the active control's `IsMenuKey` method will call the form's `IsShortcut` method and then the `Application.IsShortcut` method will call it again. At best this is just a waste of CPU cycles, but it also means that your handler for the form's `OnShortcut` event will fire twice in a row for keys it does not handle... However, it is possible to distinguish between the two calls by looking at the message: for the first call it will be `CN_KEYDOWN`, for the second `CM_APPKEYDOWN`.

Control finally returns to the `IsMenuKey` method and from there back to `TWinControl.CNKeyDown`. We are now on line 8 of [Listing 5](#). The next part of the processing is only performed at run-time, not for controls on a designer window in the IDE.

### 5.2.3 Parental guidance: the `CM_CHILDKEY` message

In line 10 of [Listing 5](#) the control sends a `CM_CHILDKEY` message with the key's virtual key code and self reference to itself, via the `Perform` method. `Perform` is the VCL equivalent of the API `SendMessage` call; it will pass a message to the control's window function directly. Basically, it just packages the passed parameters into a `TMessage` record and then calls the method pointer stored in the control's `WindowProc` property with this record. It returns the message record's `Result` field. If the handler for `CM_CHILDKEY` returns a value `<> 0` the key is considered handled and no further processing will be done for it.



The `CM_CHILDKEY` message is handled by `TWinControl.CMChildkey`. Since this is a message handler method like any other, component writers can replace it with their own version in their custom controls.



`TWinControl.CMChildkey` just passes the message to the parent control's `WindowProc` (if the control has a parent). If no control in the parent chain handles the message it will thus wind its way up the chain until it reaches the top level form. This of course gives us a prime intervention point on the form level: just add a handler for `CM_CHILDKEY` to the form and it will see all keys for all controls on the form that are not handled as shortcuts. This also works for controls on embedded frames or forms. The message's `lparam` contains a reference to the control that originally received the `CN_KEYDOWN` notification. The sample application has a form-level handler for `CM_CHILDKEY` and will log these messages if the appropriate entry in the Log menu is checked.



## 5.2.4 Navigation, default and cancel buttons: the CM\_DIALOGKEY message

The last part of the processing done by `TWinControl.CNKeyDown` deals with a specific subset of keys. These are tab, arrow keys, enter/return, escape, execute and cancel key. The last two actually have no physical representation on most of today's keyboards, as far as I'm aware. These keys are used to navigate between controls on a form or to trigger the default and cancel buttons, if the form has them. They are only to be used this way, however, if the active control does not want to use them itself, so the code first has to ask the control whether it wants these keys. It does that twice, in fact, using different mechanisms.



`TWinControl.CNKeyDown` first sends a `CM_WANTSPECIALKEY` message to the control via `Perform` (line 22 of [Listing 5](#)), passing the key's virtual key code in the message's `wparam`. The VCL does not handle this message on the level of `TWinControl`; it is handled by a few other control classes (`TCustomGrid`, `TCustomMaskEdit`), though. Component writers can add a message handler for this message to their own custom controls, if they need to handle the navigation keys.



Like all other unhandled messages, `CM_WANTSPECIALKEY` eventually ends up in `TWinControl.DefaultHandler` (which is virtual and can be overridden by component writers), which will pass it to the control's API-level default window function.

For all controls based on Windows standard or common controls this will be a function registered for this control class in Windows. For custom controls it is `DefWindowProc`, a generic window function provided by the API. A `TWinControl` stores a pointer to the window function when its window handle is created (in `TWinControl.CreateWnd`). This default window function will in fact do most of the work for the standard controls.



If `CM_WANTSPECIALKEY` returns unhandled the code (line 23 of [Listing 5](#)) sends a `WM_GETDLGCODE` message to the control. This is the message Windows controls expect to get to allow them to specify the kinds of keys they are interested in. Again the VCL's core control classes do not handle this message, but component writers can of course add a message handler for it to their own controls. In fact, they have to if their control does not descend from a standard Windows control and they want to handle text input via the keyboard. All standard Windows controls handle the message if they have a keyboard interface.



If the control does not express an interest in the key it finally gets sent as a `CM_DIALOGKEY` message to the control's parent form (line 24 and 25 of [Listing 5](#)). Of course this gives us another intervention point, since we can add a handler for this message to our form (as well as to custom controls we write). The sample application does this and will log the message if the appropriate entry in the Log menu is checked. A handler for this message is a good place to implement moving to the next input menu control via the Enter/Return key, for example. For a code example see this [newsgroup post](#). Keep in mind that this message will only be sent for a small subset of all keys!

`TCustomForm.CMDialogKey` handles the navigation between controls on the form via Tab, Shift-Tab, or the arrow keys. For others it calls the inherited handler, which turns out to be `TWinControl.CMDialogKey`. This message handler method contains only a single line of code:

```
Broadcast(Message);
```

The `Broadcast` method sends the message to all immediate children of the control (all elements of the `Controls` array), until it runs out of children or one of them returns a message result `<> 0` to indicate it has handled the message. This way the message can trickle down the children tree, eventually reaching every control on the form, regardless of how deep it is nested, including controls on frames and embedded forms. This is the mechanism `TButton` etc. use to detect and react to Enter and Escape keys even if they do not have the focus at the moment. Look at the source for `TButton.CMDialogKey` to see how this works.



There is a problem here due to the **sequence** of the traversal: if a form contains multiple buttons marked as default or cancel (e.g. on nested frames or forms) the first button found may trigger and not the one the user expects to trigger. If you have such a situation, handle the `CM_DIALOGKEY` message on the form level to make sure the frame/form containing the current active control sees the message first.

Note that `TWinControl.CMDialogKey` does not call the inherited message handler. This means the message will never reach the `DefaultHandler` method and not be passed to the default window function. This is OK since the function would not know what to do with it anyway.

## 5.2.5 Back to the message loop

Execution flow will now exit the `TWinControl.CNKeyDown` method and make its way up the call stack until it reaches `TApplication.ProcessMessage` again (see [Listing 3](#)). If nobody handled the key up to this point it will finally be passed as the original `WM_KEYDOWN` message to `DispatchMessage`, which passes it to the target control's window function. It will end up in `TWinControl.MainWndProc`, get passed to the method pointer in the `WindowProc` property and so on. Does the VCL perform any default processing for this message? You bet it does, and we'll see in a moment what that means.

## 5.3 Handling WM\_KEYDOWN

The default handler for this message is `TWinControl.WMKeyDown`. Just for completeness' sake I'll mention here that this is of course an intervention point for component writers, since they can replace this message handler. There is another and usually better intervention point coming up below, though. The handler does not in fact do much; its main action is this one:

```
if not DoKeyDown(Message) then inherited;
```

The meat is inside `DoKeyDown`; it implements the VCL's key preview feature.

### 5.3.1 Key preview on the form level

The implementation of `TWinControl.DoKeyDown` has changed a bit from Delphi 7 to Delphi 2007. The difference is in the handling of embedded forms: in Delphi 7 only the top level form's key preview was invoked, in Delphi 2007 the immediate parent form is consulted first and the top level form's key preview is only called if the immediate parent form did not mark the key as handled. Note that this only assumes one level of nesting; if an embedded form has another embedded form as child the `OnKey` events of the first embedded form will never fire if a control on the second form has the focus.

The key preview feature works like this:

- `DoKeyDown` tries to find the immediate parent form of the control, if it is not a form already.
- If it has one and the form's `KeyPreview` property is true the form's `DoKeyDown` method is called. If it returns true further processing is aborted.
- `DoKeyDown` is in fact not virtual, so `TWinControl.DoKeyDown` gets called again on the form instance.
- If the form does not have the `csNoStdEvents` flag set in its `ControlStyle` property the `KeyDown` method is called.
- `KeyDown` fires the `OnKeyDown` event.
- If there is a handler for this event and if it sets the passed `Key` parameter to 0 `DoKeyDown` returns true and further processing of the key is aborted.

- If this is not the case then the whole exercise is repeated with the top level parent form (if it is not identical to the immediate parent form).

If the preview does not mark the key as handled the original control's `KeyDown` method is called and will fire the control's `OnKeyDown` event (unless it has the `csNoStdEvents` flag set). There are two intervention points in this sequence:



The `KeyDown` method is a dynamic method and can be overridden by component writers. This is the preferred method to handle non-character keys in custom controls. Characters should be handled in `KeyPress`. Only handle `WM_KEYDOWN` if the control should not support the form key preview.



Application programmers can handle the `OnKeyDown` event either on the form or control level.

If the key is not handled in the `OnKeyDown` event of a form or control the message will eventually be delivered to the control's API window function, which will handle it in the appropriate way for the standard Windows controls. Note that handling a key in `OnKeyDown` will **not** prevent a subsequent `WM_CHAR` or `WM_KEYUP` message for the same key to be delivered.

The sample application handles the `OnKeyDown` event for the form and `edit1`. Enable the log for these through the Log menu, as usual.

## 5.4 Handling `CN_SYSKEYDOWN` and `WM_SYSKEYDOWN`

When a `WM_SYSKEYDOWN` message is retrieved from the message queue `TApplication.IsKeyMsg` will send a corresponding `CN_SYSKEYDOWN` message to the target control or the control having the mouse capture. The standard handling for this message is a scaled-down version of the handling for `CN_KEYDOWN`. The edited code for the `TWinControl.CNSysKeyDown` method is shown in [Listing 7](#) below.

```
01: begin
02:   Message.Result := 1;
03:   if IsMenuKey(Message) then
04:     Exit;
05:   if not (csDesigning in ComponentState) then
06:     begin
07:       if Perform(CM_CHILDKEY, Message.CharCode, Integer(Self)) <> 0 then
08:         Exit;
09:       if GetParentForm(Self).Perform(CM_DIALOGKEY, Message.CharCode,
10:         KeyData) <> 0
11:       then
12:         Exit;
13:     end; {if}
14:     Message.Result := 0;
15:   end;
```

**Listing 7:** `TWinControl.CNSysKeyDown`

If you compare this listing with [TWinControl.CNKeyDown](#) you'll see that they have a lot in common, so practically everything said in [chapters 5.2.2](#) to [5.2.4](#) applies here as well. For the VCL programmer both messages are handled the same way, with the same intervention points. There is only one difference of note: if the Alt key is held down the `CM_DIALOGKEY` message will be sent for all keys, not only for the navigation keys. If you need to detect whether Alt is down or not in your message or event handlers you can use the API function `GetKeyState`, like this:

```
if GetKeyState(VK_MENU) < 0 then
{ Alt is down }
```

The VCL also handles `WM_SYSKEYDOWN` exactly as it does `WM_KEYDOWN`: it calls `TWinControl.DoKeyDown` for it and thus invokes the form's key preview and fires the control's `OnKeyDown` event. There is no difference from the sequence detailed in [chapter 5.3](#).

## 5.5 Handling `CN_CHAR` and `WM_CHAR`

When a `WM_CHAR` message is retrieved from the message queue, `TApplication.IsKeyMsg` will send a corresponding `CN_CHAR` message to the target control or the control having the mouse capture. The standard handling for this message is a lot simpler than that for `CN_KEYDOWN`, but it includes one important task: handling accelerators. Accelerators, in Windows lingo, are a special type of shortcut, marked by underlined letters in control and menu captions and usually invoked by the matching `Alt-<letter key>` key combination. As we'll see in a moment, `Alt` is not always required, though.



The edited code for the `TWinControl.CNChar` method is shown in [Listing 8](#) below. Since it is a standard message handler, component writers can replace it in their custom controls.

```
01: begin
02:   Message.Result := 1;
03:   if not (csDesigning in ComponentState) then
04:   begin
05:     if ((Perform(WM_GETDLGCODE, 0, 0) and DLGC_WANTCHARS) = 0) and
06:       (GetParentForm(Self).Perform(CM_DIALOGCHAR, Message.CharCode,
07:                                     Message.KeyData) <> 0)
07:     then
08:       Exit;
09:   end; {if}
10:   Message.Result := 0;
11: end;
```

**Listing 8:** `TWinControl.CNChar`

The message handler asks the current control whether it wants to get character messages through a `WM_GETDLGCODE` message. If it declines, the character is sent as a `CM_DIALOGCHAR` message to the top level form on which the control sits. Do you remember that I said a couple of [pages earlier](#) that custom controls that want to handle keyboard input must handle the `WM_GETDLGCODE` message? Here is the reason why: if they do not set the `DLGC_WANTCHARS` bit in the message's return value they will never get any `WM_CHAR` messages for characters used as accelerators on the form.

There is another point of note: if the current control does not handle character input a character will act as an accelerator **even if the `Alt` key is not down!** If you want to prevent this you can handle the message on the form level as shown in this [newsgroup post](#). If `Alt` is down the generated character message will be `WM_SYSCHAR` anyway, not `WM_CHAR`. We'll get to that message further down.

### 5.5.1 Accelerator handling: the `CM_DIALOGCHAR` message

`CM_DIALOGCHAR` is handled very similarly to `CM_DIALOGKEY`. The default handler in `TWinControl` (`TCustomForm` does not replace it) just broadcasts the message to its immediate children. It will thus trickle down the control tree until it finds a control that handles it as a shortcut and returns a value `<> 0` as message result, or all controls have been visited. This traversal includes controls on nested frames and forms as well. Look at the source code for `TButton.CMDialogChar` or `TCustomLabel.CMDialogChar` to see how handling this message works.



There is a problem here due to the **sequence** of the traversal: if a form contains multiple controls using the same accelerator (e.g. on nested frames or forms), the first control found may handle the accelerator, instead of the one the user expects to handle it. If you have such a

situation, handle the `CM_DIALOGCHAR` message on the form level to make sure the frame/form containing the current active control sees the message first.



Component authors need to handle the `CM_DIALOGCHAR` message in their custom controls if they want to support accelerators. Application programmers can handle the message on the form level to redirect the message to a nested frame or form, for example. See this [newsgroup post](#) for a way to fix the problem mentioned in the previous paragraph.

## 5.5.2 Character preview: the `WM_CHAR` message

The VCL's default handling of the `WM_CHAR` message is quite similar to the one for `WM_KEYDOWN`. The message handler in `TWinControl` (which component writers can replace in their custom controls) calls the `DoKeyPress` method and passes the message on to the inherited handler if `DoKeyPress` does not mark it as handled. This way it will eventually end up at `TWinControl.DefaultHandler` and get passed to the control's API window function.



`DoKeyPress` first checks if the control's top level parent form has its `KeyPreview` property set to true. If this is the case the form's `DoKeyPress` method is called, which will fire the form's `OnKeyPress` event. This is of course the preferred intervention point on the form level for application programmers for character processing. The event handler can set the passed character to `#0` to mark it as handled.



Note that there is an inconsistency here in the Delphi 2007 implementation: in contrast to `DoKeyDown`, the `DoKeyPress` method only fires the matching event of the top-level form on which the control sits; it will not handle embedded forms correctly by firing the immediate parent form's `OnKeyPress` first.



`DoKeyPress` calls the `KeyPress` method if the character was not handled by the form's `OnKeyPress` handler. This is a dynamic method component writers can override in their custom controls to implement character processing. The method only fires the control's `OnKeyPress` event, which is of course the application programmer's hook into a control's character processing.

## 5.6 Handling `CN_SYSCHAR` and `WM_SYSCHAR`

As with the key down messages, the handling of the `CN_SYSCHAR` and `WM_SYSCHAR` messages is very similar to that of their non-SYS equivalents. A `CN_SYSCHAR` message will send a `CM_DIALOGCHAR` message to the top-level form on which the control sits, unless the key pressed is the spacebar. Alt-`<spacebar>` is a system shortcut to drop down a window's system menu, so a control should not be allowed to handle it differently. As detailed in [chapter 5.5.1](#), the `CM_DIALOGKEY` message is the VCL's mechanism to handle accelerators. This time they will be Alt-`<letter key>` combinations.

The VCL does not handle the `WM_SYSCHAR` message at all in the base control classes.

## 5.7 Handling `CN_KEYUP` and `WM_KEYUP`

Things get simpler as we get further downstream in the key processing cascade. The `TWinControl.CNKeyUp` method just sends a `CM_WANTSPECIALKEY` message to the target control for the same set of keys that `TWinControl.CNKeyDown` sends this message for: the tab, arrow keys, enter/return, escape, execute and cancel keys. If the key message processing did not result in a change of focus between key down and key up, this means the same control will receive this message twice, with the exact same parameters. If the message handler needs to distinguish these two it has to check the key's current state via `GetKeyState` to see if the key is currently down or up again. If the focus moved between key down and key up, each control involved will get the message once. You can see this in action in the sample application if you enable the appropriate log menu item.



The processing of `WM_KEYUP` follows the same pattern we have already seen for `WM_KEYDOWN` and `WM_CHAR`: the default message handler in `TWinControl` calls the `DoKeyUp` method, and then this one calls the top-level form's `DoKeyUp`, if the form has `KeyPreview` set to true, which fires the form's `OnKeyUp` event. If the event handler does not declare the key handled by setting it to 0, `KeyUp` is called (again a dynamic method component authors can override), which then fires the control's `OnKeyUp` event. This again gives the application programmer opportunities to trap the key up on the form or control level.



Applications should normally not handle key up messages or events since the behavior may not always be what the programmer intended. Most processing triggered by keyboard input is done in key down or key press message or event handlers. Using key up as a trigger suffers from one major problem: the control receiving the message may do so more or less by accident. Consider this scenario:

With focus sitting on control A on a form the user invokes a modal dialog using a main menu item. The dialog has default and cancel buttons. The user presses the Enter key, the default button triggers on the key down message and the dialog closes. Where will the key up message for the Enter key go now? To whatever control gets the focus back after the dialog has closed. If that control processes `VK_RETURN` in its `OnKeyUp` handler some unintended action will result.

## 5.8 Handling `CN_SYSKEYUP` and `WM_SYSKEYUP`

Here it gets even simpler: the VCL does not handle `CN_SYSKEYUP` in the base control classes (in fact there is no message with this name declared in the Controls unit) and `WM_SYSKEYUP` is handled exactly like `WM_KEYUP`: it calls the `DoKeyUp` method and thus fires the form's and control's `OnKeyUp` events eventually, via `KeyUp`.

## 6. Summary

This brings us to the end of the journey. If you managed to hold on up to the finish you are now entitled to an extra donut <g>. It is a bit daunting to realize that a simple keystroke will execute hundreds and hundreds of lines of code in the VCL alone, not counting what goes on under the hood in Windows. On the other hand, the design used is very flexible and offers a lot of intervention points for the programmer to customize key processing to his applications' requirements.

The following outline tries to condense the information detailed in the preceding chapters for quick reference. The nesting level reflects the call depth approximately. Keep in mind that processing can be aborted at nearly any step by marking a message or key as handled.

1. The user presses a key on the keyboard.
2. The system places a `WM_KEYDOWN` or `WM_SYSKEYDOWN` message into the foreground window's thread's message queue.
3. The thread's message loop fetches the message from the queue (`Application.ProcessMessage`).
  1. The `Application.OnMessage` event is fired with the message.
  2. `Application.PreProcessMessage` is called and calls the target control's `PreProcessMessage` method.
  3. `Application.IsHintMessage` is called and calls the active hint window's `IsHintMsg` method.
  4. `Application.IsMDIMsg` is called.

5. `Application.IsKeyMsg` is called and sends a `CN_KEYDOWN` or `CN_SYSKEYDOWN` message to the target control or the control having the mouse capture.
  1. `TWinControl.CNKeyDown` (or `CNSysKeyDown`) receives the message.
  2. The method calls `TWinControl.IsMenuKey`.
    1. `IsMenuKey` ask the control's popup menu if it wants to handle the key. This procedure is repeated for the controls up the parent chain until the top form is reached.
    2. The parent form's `IsShortcut` method is called.
      1. The form's `OnShortcut` event is fired.
      2. The components on the form are enumerated recursively and each action list found is asked whether it wants to handle the key as shortcut.
    3. A `CM_APPKEYDOWN` message is sent to the Application window.
      1. The `Application.WndProc` method receives the message.
      2. The message is passed to any hooks established via `Application.HookMainWindow`.
      3. The `y` method is called.
        1. The `Application.OnShortcut` event is fired.
        2. The main form's `IsShortcut` method is called, fires the main form's `OnShortcut` event and passes the key to any actions lists owned by the form, recursively.
  3. A `CM_CHILDKEY` message is sent to the control and will work its way up the parent chain to the form level.
  4. If the key is a navigation key (tab, arrows, enter, esc)
    1. A `CM_WANTSPECIALKEY` message is sent to the control.
    2. A `WM_GETDLGCODE` message is sent to the control.
    3. A `CM_DIALOGKEY` message is sent to the top level form. If the key is not tab or an arrow key the message will be sent to all child controls recursively.
6. `Application.IsDlgMsg` is called.
7. `TranslateMessage` is called and will create a `WM_CHAR`, `WM_SYSCHAR`, `WM_DEADCHAR` or `WM_SYSDEADCHAR` message for keys that create characters.
8. `DispatchMessage` is called.
  1. The `TWinControl.WMKeyDown` (or `WMSysKeyDown`) method receives the message.
  2. `TWinControl.DoKeyDown` is called.
    1. If the control's parent form has `KeyPreview` enabled the form's `DoKeyDown` method will be called.
      1. The form's `KeyDown` method is called and fires the `OnKeyDown` event of the form.
    2. If the form's immediate parent form is nested (docked) in another form the process is repeated for the top level form.



3. The control's `KeyDown` method is called and fires the control's `OnKeyDown` event.
3. `TWinControl.WMKeyDown` (or `WMSysKeyDown`) passes the message to the inherited message handler, it eventually ends up in `TWinControl.DefaultHandler` and is passed to the control's API window function.
9. Control returns to the message loop.
4. If the key creates a character the thread's message loop fetches the `WM_CHAR` or `WM_SYSCHAR` (or `WM_DEADCHAR`) message from the queue (`Application.ProcessMessage`).
  1. The `Application.OnMessage` event is fired with the message.
  2. `Application.PreProcessMessage` is called and calls the target control's `PreProcessMessage` method.
  3. `Application.IsHintMessage` is called and calls the active hint window's `IsHintMsg` method.
  4. `Application.IsMDIMsg` is called.
  5. `Application.IsKeyMsg` is called and sends a `CN_CHAR` or `CN_SYSCHAR` message to the target control or the control having the mouse capture.
    1. `TWinControl.CNChar` (or `CNSysChar`) receives the message.
    2. The method sends a `WM_GETDLGCODE` message to the control to see if it wants to process character input (only for `CNChar`).
    3. A `CM_DIALOGCHAR` message is sent to the top level form. The message will be sent to all child controls of the form, recursively .
  6. `Application.IsDlgMsg` is called.
  7. `TranslateMessage` is called but does nothing in this case.
  8. `DispatchMessage` is called.
    1. The `TWinControl.WMChar` (`WMSysChar` does not exist) method receives the message.
    2. `TWinControl.DoKeyPress` is called.
      1. If the control's parent form has `KeyPreview` enabled the form's `DoKeyPress` method will be called.
        1. The form's `KeyPress` method is called and fires the `OnKeyPress` event of the form.
      2. The control's `KeyPress` method is called and fires the control's `OnKeyPress` event.
    3. `TWinControl.WMChar` passes the message to the inherited message handler, it eventually ends up in `TWinControl.DefaultHandler` and is passed to the control's API window function. `WM_SYSCHAR` ends up in `DefaultHandler` directly since there is no message handler method for it.
  9. Control returns to the message loop.
5. The thread's message loop fetches the `WM_KEYUP` or `WM_SYSKEYUP` message from the queue (`Application.ProcessMessage`).
  1. The `Application.OnMessage` event is fired with the message.

2. `Application.PreProcessMessage` is called and calls the target control's `PreProcessMessage` method.
3. `Application.IsHintMessage` is called and calls the active hint window's `IsHintMsg` method.
4. `Application.IsMDIMsg` is called.
5. `Application.IsKeyMsg` is called and sends a `CN_KEYUP` or `CN_SYSKEYUP` message to the target control or the control having the mouse capture.
  1. `TWinControl.CNKeyUp` (`CNSysKeyUp` does not exist) receives the message.
  2. The method sends a `CM_WANTSPECIALKEY` message to the control if the key is one of the navigation keys.
6. `Application.IsDlgMsg` is called.
7. `TranslateMessage` is called but does nothing in this case.
8. `DispatchMessage` is called.
  1. The `TWinControl.WMKeyUp` (or `WMSysKeyUp`) method receives the message.
  2. `TWinControl.DoKeyUp` is called.
    1. If the control's parent form has `KeyPreview` enabled the form's `DoKeyUp` method will be called.
      1. The form's `KeyUp` method is called and fires the `OnKeyUp` event of the form.
      2. The control's `KeyUp` method is called and fires the control's `OnKeyUp` event.
    3. `TWinControl.WMKeyUp` (or `WMSysKeyUp`) passes the message to the inherited message handler, it eventually ends up in `TWinControl.DefaultHandler` and is passed to the control's API window function.
9. Control returns to the message loop.
6. If no more messages are waiting in the queue `Application.Idle` is called and blocks the main thread until a new message comes in.

## Appendix 1

### How does the VCL find a control from its window handle? How are control objects and window handles tied together?

As detailed further up, the [IsKeyMsg](#) method called from the message loop tries to figure out if the window handle it has for the message target belongs to a VCL control. If not it walks up the window parent chain to find a VCL control. How do you get from a window handle to a control reference? The other way around is trivial: just use the `Handle` property any `TWinControl` descendent inherits...

The VCL has a number of functions related to the task of finding a control reference from its window handle. These are `IsVCLWindow` and `FindControl` in the Controls unit. The former just calls the latter and checks if a control reference is returned.

To understand how `FindControl` works we first have to investigate how the VCL ties a `TWinControl` to a window handle. The Windows API offers several different mechanisms to tie application-specific data to a window handle.

A window is an instance of a class (the window class, similar to how objects in Delphi are instances of classes). A class has to be registered with Windows through a call to `RegisterClassEx`, for

which the programmer provides a data record with information that defines the behavior of the class and the windows created from it. Part of this is the class's window function, but the programmer can also request additional memory to be added to Windows' internal representation of the class and also to each instance of the windows created for this class. A window class itself, as well as any window, is represented by a block of memory in Windows, and you can think of the window handle as an (opaque) reference to the window's data block. This additional memory can then be accessed through API functions like `SetClassLong` and `SetWindowLong` and the corresponding get functions. The problem with this mechanism is that it is accessible to anyone that has a window handle available, access to the memory is by index (the additional memory is treated as an array of `DWORD`) and anybody (mis)using these functions could mess up information stored this way.

The second mechanism available is **window properties**. Each window can have a list of properties associated with it (see `SetProp` and `GetProp` API functions). Properties are identified using string names (with a twist, see below) and each property can store a `HANDLE`-sized data item (32 bit in 32 bit Windows, 64 bit in 64-bit Windows). That is exactly what is needed to store an object reference. And if the string name is not known to the outside world and is reasonably uncommon the chance of a programmer using window properties that collide with the VCL's use of them is very small. So the VCL uses window properties to store the control reference with a window handle.

If you look at the `InitControls` procedure in the Controls unit (it is called during unit initialization) you can see how the string names for the properties used by the VCL are set up:

```
WindowAtomString := Format('Delphi%.8X', [GetCurrentProcessID]);
WindowAtom := GlobalAddAtom(PChar(WindowAtomString));
ControlAtomString := Format('ControlOfs%.8X%.8X', [HInstance,
GetCurrentThreadId]);
ControlAtom := GlobalAddAtom(PChar(ControlAtomString));
RM_GetObjectInstance := RegisterWindowMessage(PChar(ControlAtomString));
```

The variables assigned are unit-level variables declared near the start of the unit's Implementation section. As you can see, the first of the names generated encodes the current process ID; the second encodes the module handle and the thread ID (of the main thread in a VCL forms application, since that thread will initialize the unit). What are these atom thingies, though?

Windows contains a kind of global hash table for string values. This is called the global atom table. Adding a string to the table via `GlobalAddAtom` returns a 16-bit hash code for the string, called an atom. For efficiency reasons, window properties do not store the string name as such; they store the associated atom. Both `SetProp` and `GetProp` can be called either with a string (`PChar`) or an atom. The latter is faster since the function does not need to look up the string in the global atom table first. So the VCL obtains the atoms up front. Also note that a custom windows message is registered using one of the strings generated. This is used in a fallback mechanism introduced in Delphi 6. We'll get to that later.

Where are the window properties for a control's `Handle` set? This happens directly after the window handle has been created. This process is a wee bit convoluted. The VCL needs a way to redirect messages send to the window through its handle to a method of the control object that is bound to the handle. But a window's window function has to have a specific signature that is not compatible with a method, due to the hidden `Self` parameter every method has. To solve this problem, the VCL creates a window function on the fly, hardcoding the control's object reference and the necessary code to rearrange the parameters coming from Windows as well as the address of the method to call in the generated code (in fact the stub calls `Classes.StdWndProc` for some of the work). This is done by the `MakeObjectInstance` function (Classes unit), which is called in the `TWinControl` constructor.

A `TWinControl` will create its window handle on an as needed basis. It calls the `TWinControl.CreateWnd` method for this purpose (a virtual method descendants can override). `CreateWnd` calls `CreateParams` (also virtual) to fill a record with the information necessary to

register the window class, checks if the class is already registered, and if not, registers it. At this step it uses a specific window function (`InitWndProc`) for the registration. The next step then creates the actual window handle:

```
CreationControl := Self;  
CreateWindowHandle(Params);
```

`CreationControl` is a unit-level (global) variable, `CreateWindowHandle` a method of `TWinControl` that calls the `CreateWindowEx` API function. The first message Windows sends to the new window (`WM_NCCREATE`, I think) is received by `InitWndProc`. The function stores the window handle received as message parameter into `CreationControl.FHandle`, then subclasses the window in the API way (via `SetWindowLong`) to make the code stub created via `MakeObjectInstance` the new window function, and then finally sets the two window properties:

```
SetProp(HWindow, MakeIntAtom(ControlAtom), THandle(CreationControl));  
SetProp(HWindow, MakeIntAtom(WindowAtom), THandle(CreationControl));
```

As you see, it stores the control's object reference as the value of both properties. The last thing done is to pass the received message to the new window function (which happens to call `TWinControl.MainWndProc`) and to clear the `CreationControl` variable.

Why is this process implemented in such a complex manner? Why not simply use the `MakeObjectInstance`-generated stub address for the window function directly instead of going through `InitWndProc`, and store the window handle returned by `CreateWindowEx`?

The original designers of the VCL got themselves trapped here due to a particular design decision: the `TMessage` record containing the message parameters for a `TWndMethod` (the type of method `MakeObjectInstance` can hook to a stub) does not contain the message's target window handle. That makes sense in general since the control has the window handle available through its `Handle` property. More importantly, it is also required due to the way the VCL's message dispatching (`TObject.Dispatch`) works. The first field of the message record passed to `Dispatch` has to be the message ID.

The crux is that the control's `Handle` would not yet be set for messages sent to the window before `CreateWindowEx` has returned and so there would be no way to send messages received during this period to the default window function. And that would foul things up since Windows sends some messages during window initialization that must be passed to the default window function. So we are stuck with this complicated sequence and also have to live with the fact that it is not thread-safe, due to the use of global variables.

OK, now we know enough to understand how `FindControl` works. The relevant code is shown in [Listing 9](#) below.

```
function FindControl(Handle: HWND): TWinControl;  
var  
    OwningProcess: DWORD;  
begin  
    01: Result := nil;  
    02: if (Handle <> 0)  
    03:     and (GetWindowThreadProcessID(Handle, OwningProcess) <> 0)  
    04:     and (OwningProcess = GetCurrentProcessId)  
    05: then begin  
    06:     if GlobalFindAtom(PChar(ControlAtomString)) = ControlAtom then  
    07:         Result := Pointer(GetProp(Handle, MakeIntAtom(ControlAtom)))  
    08:     else  
    09:         Result := ObjectFromHWND(Handle);  
    end;  
end;  
end;
```

**Listing 9:** FindControl

This function has seen some extensions since Delphi 5 to add checks for some corner cases. The function first checks if the window handle is valid and belongs to a window in the current process (lines 2 to 4). Then it checks whether the `ControlAtom` is still intact. The global atom table gets cleaned when the current user logs off, so this can cause problems for services (which should not be using windowed controls anyway, though). If the atom is still valid the code then reads the matching window property to get the stored object reference. If the atom is not valid any more the fallback mechanism is invoked. `ObjectFromHWnd` sends the registered `RM_GetObjectInstance` message to the window. The handler for that message in `TWinControl` returns the object's self reference.



There is a snag one should be aware of here. The `ControlAtom` string encodes both the current module handle (`HInstance`) and the main thread ID. "Current module" in this case is the module containing the Controls unit. For a normal VCL-based application (EXE) not built with run-time packages, this will be the exe's module handle itself. For an executable built with run-time packages it will be the module handle of the core VCL package. For a DLL not built with run-time packages it will be the DLL's module handle. For a DLL used by an EXE, all modules built without run-time packages, this means that the host application's message loop **will not recognize TWinControls created in the DLL as VCL controls!** `FindControl`, executed in the host application for a window handle belonging to a control created in the DLL, will return nil. Fortunately `TApplication.IsKeyMsg` will send its `CN_*` notifications to the DLL windows anyway, so the key preprocessing will work for modeless DLL forms.

In case you are wondering: the window property tied to `WindowAtom` is used by the VCL drag & drop support to figure out if the window under the cursor is a VCL control belonging to the same process.

## Appendix 2

### How does message passing via Dispatch work in Delphi?

The VCL has a message-passing mechanism built into the `TObject` class. A message is sent to the object by calling the `Dispatch` method with an arbitrary data block (the method has an untyped parameter). The first word of this data block is interpreted as a message ID and this ID is used to find a handler method for the message. If none is found the message is handed to the `DefaultHandler` method. How are message IDs tied to handler methods?

Let's quickly recap the kinds of methods an object can have. Not counting class methods and constructors (which are invoked on a class reference and not on an object) we have:

#### Static methods

The address of a static method is known at compile time and a call to the method is coded using the address directly. A descendant class cannot replace a static method of an ancestor class polymorphically.

#### Virtual methods

These are methods a descendant can override. A class has an associated **virtual method table** (VMT) which contains the addresses of the class's virtual methods. Each class contains a full copy of the VMT of its immediate ancestor class, which is extended with new slots for new virtual methods introduced in the class. For overridden methods the new method address replaces the method address of the ancestor in the VMT. Calls to virtual methods are implemented as indirect calls through the classes VMT. The compiler knows the index of the method to call, so the call is efficient. Since each VMT is an extended copy of the ancestor class's VMT, deep class hierarchies with many virtual methods consume significant memory for the VMTs alone. Destructors are a special kind of virtual method.

#### Dynamic methods

These are also methods a descendent can override. They are implemented differently from

virtual methods, though. A class having dynamic methods has a **dynamic method table** (DMT).

In contrast to the VMT, the DMT is not an integral part of the class record. The class record has a field that contains a pointer to the DMT. The DMT itself starts with a word containing the number of entries in the DMT (Count). This is followed by Count words containing the **selector values** for the dynamic methods. This in turn is followed by Count pointers holding the dynamic method addresses.

Finding a dynamic method using its selector as input is the task of the `GetDynaMethod` function in the System unit. It conducts a sequential search over the class's DMT. If the selector is not found there it repeats with the ancestor classes' DMTs until it either finds a match or ends up at TObject. Unlike VMTs, a class's DMT contains only the dynamic methods introduced or overridden in the class. This saves space (a real concern at the time Delphi 1 came out) but at the price of performance, since finding a method in a DMT is much slower than a virtual method call through the VMT.

For methods declared with the `dynamic` keyword the selector value (message ID) is picked by the compiler. Dynamic methods must use the register calling convention (the default).

### Message handler methods

These are methods declared with the `message` keyword. Message handler methods are just dynamic methods with the selector values specified by the programmer. They are found through the class's DMT, just like dynamic methods are. There is a restriction, though: a message handler method must have a single var parameter and use the register (default) calling convention.

What `Dispatch` gives us is simply a generic way to call a dynamic method. The first two bytes of the parameter passed contain the selector for the method. `Dispatch` calls `GetDynaMethod` to find the address for the dynamic method and calls it; passing the parameter it received itself (by reference).



There is one important point to keep in mind: the compiler cannot check if a message handler method is declared in a way that is compatible with the parameter passed to `Dispatch`. There is no type safety in this low-level area. It is the responsibility of the programmer to make sure the data passed to `Dispatch` does match the format expected by the handler method. For all message handler methods declared in the VCL this means that the message record used has to be at least `Sizeof(TMessage)` bytes in size and follow the same layout for the first four dwords. Message handler methods for Windows or VCL messages often write to `Message.Result`, and that is the fourth dword in the record. Passing smaller records to such a handler via `Dispatch` will result in memory corruption or access violations.