Manufacturing Fake Keystrokes

by Brian Long

Here's another question originally posed by a reader of *The Delphi Clinic*, but expanded into an article because of the volume of information to be covered:

I have seen information on the internet that allows me to manufacture keypresses programmatically and have understood the basic principles. However, I am having trouble emulating the Alt+NumPad *keys to generate extended characters. I am currently attempting to use the* KeyBd_Event *function from the Windows API. I am stuck. Can you enlighten me?*

When people look into faking keystrokes programmatically, they often walk straight into a trap. When keys are pressed, messages are generated, such as wm_KeyDown, wm_KeyUp and wm_Char (see *Keystroke Interception* in Issue 38's *Delphi Clinic* for substantially more detail on this subject). Given this information, the impulsive action is to simply send these messages to the target window and hope for the best. Generally, this does not do the job, as Jeffrey Richter emphasised in an article he wrote in a 1992 issue of *Microsoft Systems Journal* (Volume 7, Number 8). Windows seems to want more than just these messages, which are only the end result of a keypress. Typically, this involves setting internal key state information.

In 16-bit Windows, you can jump through some hoops involving a window journal playback hook and, as is the nature of hooks, it can get quite nasty. You can also take advantage of a nice routine called PostVirtualKeyEvent, supplied in the PenWindows DLL that you may or may not have installed. Evidently this is not ideal since you might not have the DLL available, but it is discussed in Chapter 15 of *The Revolutionary Guide To Delphi 2*. We will come back to 16-bit Windows later.

In 32-bit, we have a ready-made solution in the already mentioned KeyBd_Event API. This fakes a real keypress, including the internal key state information. Of course, it also subsequently generates the expected keyboard messages as well. The idea is to make sure the target window has focus and then use KeyBd_Event to manufacture the various keypress and release actions as required.

The declaration of this routine in the Windows unit looks like this:

```
procedure KeyBd_Event(
  bVk: Byte; bScan: Byte;
  dwFlags, dwExtraInfo: DWORD);
  stdcall;
```

The first parameter is supposed to be the virtual key code for the key being pressed. There is some discussion of virtual key codes back in Issue 38's *Delphi Clinic* on page 59.

The second parameter is the key's hardware scan code, as generated by the BIOS. Some old PC technical references will have information about which keys generate which scan codes, but we don't need to worry about that. Instead we can call MapVirtualKey, which can take a virtual key code and will manufacture the corresponding scan code. The only exception to this rule is with the Print Screen key. This key can be used to copy the whole desktop as a bitmap onto the clipboard (Print Screen) or copy the active window onto the clipboard (Alt+Print Screen). If you pass the Print Screen virtual key code (vk_SnapShot) as the first parameter, then you should pass 1 as the scan code for a full screen snapshot or 0 for a window snapshot.

The third parameter specifies extra flags, including information to distinguish whether the key is being pressed (0) or released (KeyEventF_KeyUp), or whether it was an 'extended' key (KeyEventF_ExtendedKey). The BIOS generates two scan codes for extended keys, the first one always being $E0 or 224. Extended keys include Page Up, Page Down, Home, End, the cursor keys, Insert and Delete, both Windows keys and the context menu key amongst others. Fortunately, though, it seems not particularly important whether we indicate if a key is extended or not.

The fourth parameter seems safe to ignore for most purposes, but in truth allows you to send arbitrary extra information with a message. An application can extract the extra information relating to the last message plucked from the message queue with the GetMessageExtraInfo API. Usually, only the mouse or keyboard driver would add this information, but since the keyboard driver manufactures keyboard messages using KeyBd_Event, we can do likewise if we wish. Incidentally, the mouse driver manufactures mouse messages using a similar Mouse_Event API call that you may like to explore.

So that's the basics about KeyBd_Event. *The Delphi Magazine* has had mention of this API before. Octavio Hernadez used it in a submission to the old *Tips & Tricks* column in Issue 26. Before continuing I should mention that Windows 98 and Windows NT 4.0 Service Pack 3 introduce another way of manufacturing mouse and keyboard messages using the SendInput API (added to the Windows import unit in Delphi 4).

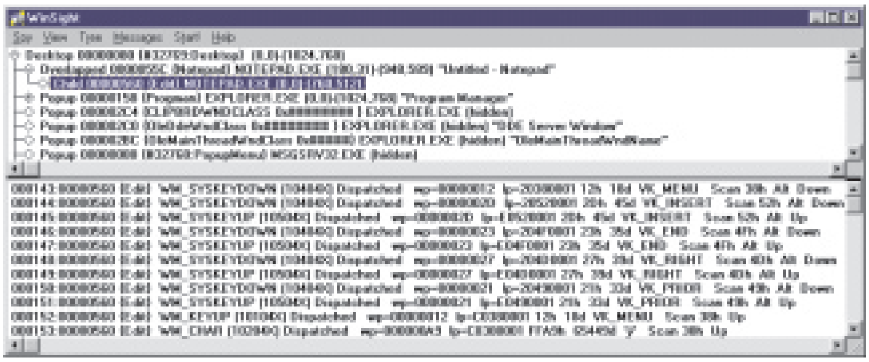
Also, going back to this 16-bit Windows thing, I mentioned that it typically involves Windows hooks. Whilst this is generally true, we can also access KeyBd_Event in 16-bit, but it is defined to take no parameters, they have to be passed in CPU registers (yikes!). So all in all it looks like we need to investigate 16-bit KeyBd_Event, 32-bit KeyBd_Event and (the very recent) 32-bit SendInput.

We won't be looking at the journal playback hook that can potentially be used to help out here, as in 32-bit mode this is restricted to playing keystrokes back to the thread that set up the hook, which limits its usefulness. However if you wish to see code and explanation that plough through this issue, I recommend Delphi 1 users check *Delphi Developer's Guide*, by Steve Teixeira and Xavier Pacheco (SAMS Publishing), Chapter 21, p528. 32-bit programmers should check either *Delphi 2 Developer's Guide*, Chapter 23, p917, or *Delphi 4 Developer's Guide*, Chapter 13 p353, by the same authors.

OK, now back to the problem in hand. The questioner wants to mimic a Windows user manufacturing a character by holding down the `Alt` key and typing the character number on the numeric keypad.

It's worth noting at this point that `Alt` and a three digit number represents an OEM or ASCII character, whereas `Alt` and a four digit number gives an ANSI character. So, the symbol © is ASCII character 184 and ANSI character 169. This means that you can get a copyright symbol using either `Alt+0169` or `Alt+184` (so long as you are using the numeric keypad, and not normal number keys).

To see which keystrokes (or more particularly which keystroke messages) are generated by `Alt+0169`, you can enlist the help of WinSight. WinSight is a tool accessible from Delphi's program group, and was used quite a lot to get much of the information about how keystrokes manufacture messages in Issue 38's *Clinic* piece. Having launched WinSight and a copy of Notepad as a guinea pig, locate Notepad in WinSight's window tree. Expanding the top level Notepad window shows the internal multi-line edit control, which is most of what Notepad is.



```
Desktop 00000080 {#32769:Desktop} (0,0)-(1024,768)
  Overlapped 0000055C {Notepad} NOTEPAD.EXE (180,21)-(940,595) "Untitled - Notepad"
    Child 00000560 {Edit} NOTEPAD.EXE (0,0)-(748,512)
  Popup 00000158 {Progman} EXPLORER.EXE (0,0)-(1024,768) "Program Manager"
  Popup 000002C4 {CLIPBRDWNDCLASS 0x########} EXPLORER.EXE (hidden)
  Popup 000002C0 {OleDdeWndClass 0x########} EXPLORER.EXE (hidden) "DDE Server Window"
  Popup 000002BC {OleMainThreadWndClass 0x######} EXPLORER.EXE (hidden) "OleMainThreadWndName"
  Popup 00000080 {#32768:PopupMenu} MSGSRV32.EXE (hidden)

000143:00000560 {Edit} WM_SYSKEYDOWN (0104h) Dispatched  wp=00000012 lp=20380001 12h 18d VK_MENU  Scan 38h Alt Down
000144:00000560 {Edit} WM_SYSKEYDOWN (0104h) Dispatched  wp=0000002D lp=20520001 2Dh 45d VK_INSERT  Scan 52h Alt Down
000145:00000560 {Edit} WM_SYSKEYUP (0105h) Dispatched  wp=0000002D lp=E0520001 2Dh 45d VK_INSERT  Scan 52h Alt Up
000146:00000560 {Edit} WM_SYSKEYDOWN (0104h) Dispatched  wp=00000023 lp=204F0001 23h 35d VK_END  Scan 4Fh Alt Down
000147:00000560 {Edit} WM_SYSKEYUP (0105h) Dispatched  wp=00000023 lp=E04F0001 23h 35d VK_END  Scan 4Fh Alt Up
000148:00000560 {Edit} WM_SYSKEYDOWN (0104h) Dispatched  wp=00000027 lp=204D0001 27h 39d VK_RIGHT  Scan 4Dh Alt Down
000149:00000560 {Edit} WM_SYSKEYUP (0105h) Dispatched  wp=00000027 lp=E04D0001 27h 39d VK_RIGHT  Scan 4Dh Alt Up
000150:00000560 {Edit} WM_SYSKEYDOWN (0104h) Dispatched  wp=00000021 lp=20490001 21h 33d VK_PRIOR  Scan 49h Alt Down
000151:00000560 {Edit} WM_SYSKEYUP (0105h) Dispatched  wp=00000021 lp=E0490001 21h 33d VK_PRIOR  Scan 49h Alt Up
000152:00000560 {Edit} WM_KEYUP (0101h) Dispatched  wp=00000012 lp=C0380001 12h 18d VK_MENU  Scan 38h Up
000153:00000560 {Edit} WM_CHAR (0102h) Dispatched  wp=000000A9 lp=C0380001 FFA9h 65449d 'Ÿ'  Scan 38h Up
```

➤ *Figure 1*

Then you ask WinSight to trace messages going to the edit control (`Messages | Select Windows`). You can also use `Messages | Options...` to ensure only input messages are being traced, to keep all the many other messages out of the picture. Having started WinSight tracing messages (by pushing its `Start!` menu item), then switching to Notepad, pressing `Alt+0169`, and stopping WinSight's tracing (by pressing `Stop!`) you get the information shown in Figure 1.

So, `Alt` (`vk_Menu`) is pressed and held down. Then `0`, or the `Insert` key on the numeric keypad (`vk_Insert`), is pressed and released, followed by a press and release of `1`, or `End` (`vk_End`). Then comes a press and release of `6`, or the `right arrow` (`vk_Right`) and a press and release of `9` or `Page Up` (`vk_Prior`). Finally, the `Alt` key is released and something spots all of this and generates ANSI character `$A9` or © (which WinSight seems strangely to draw as Ÿ). These traced messages should make more sense if you read the *Intercepting Keystrokes* entry in Issue 38's *Clinic*.

So, to programmatically emulate this we need to map these keystrokes back into calls to `KeyBd_Event`. Listing 1 does a suitable job in Delphi 2 and above, and will effectively type `Alt+0169` in whatever control has focus, but we will not leave it at this. Instead, we will use some wrapper routines to avoid many explicit calls to `KeyBd_Event`, and also to allow conditional code to give us one source file that works in 16-bit and 32-bit Delphi.

The first generic approach can be found in KeyTest1.Dpr, supplied on the disk. This project works with Delphi 1, 2, 3 and 4 and consists of a form unit, as well as a re-usable unit containing the real code (KeysU.Pas). The form, as shown running in Figure 2, can capture the screen or current window by faking a press of the `Print Screen` button. As you can see, the checkbox is currently checked, and the button has been pressed to capture the whole of my desktop and then copy it to a `TImage` component on the form.

The first button demonstrates slightly more interesting keystroke sequences. It hunts out Delphi's About box, and then pretends to type in the various undocumented keystroke sequences that bring up the Delphi Easter Eggs (see *The Delphi Clinic* in Issue 38 for more information about these). To choose a particular Easter Egg, use the spin edit control and then push the `Delphi` button.

Finally, the third button adds a copyright symbol into an edit control, as per the questioner's requirement. In fact, to make things a bit more interesting it actually inserts the string `Copyright: ©` into the edit.

The `Print Screen` button is straightforward, and has a mere two lines in its `OnClick` event

➤ *Listing 1*

```
const
  KeyEventF_KeyDown = 0;
...
KeyBd_Event(vk_Menu,   MapVirtualKey(vk_Menu,   0), KeyEventF_KeyDown, 0);
KeyBd_Event(vk_Insert, MapVirtualKey(vk_Insert, 0), KeyEventF_KeyDown, 0);
KeyBd_Event(vk_Insert, MapVirtualKey(vk_Insert, 0), KeyEventF_KeyUp,   0);
KeyBd_Event(vk_End,    MapVirtualKey(vk_End,    0), KeyEventF_KeyDown, 0);
KeyBd_Event(vk_End,    MapVirtualKey(vk_End,    0), KeyEventF_KeyUp,   0);
KeyBd_Event(vk_Right,  MapVirtualKey(vk_Right,  0), KeyEventF_KeyDown, 0);
KeyBd_Event(vk_Right,  MapVirtualKey(vk_Right,  0), KeyEventF_KeyUp,   0);
KeyBd_Event(vk_Prior,  MapVirtualKey(vk_Prior,  0), KeyEventF_KeyDown, 0);
KeyBd_Event(vk_Prior,  MapVirtualKey(vk_Prior,  0), KeyEventF_KeyUp,   0);
KeyBd_Event(vk_Menu,   MapVirtualKey(vk_Menu,   0), KeyEventF_KeyUp,   0);
```

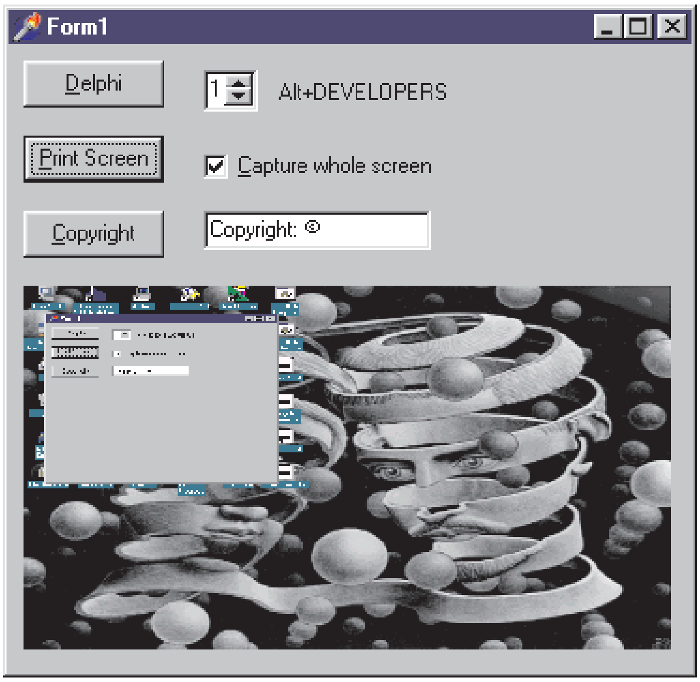
handler. You can see that the keypress is simulated using a routine called SendKeys, declared in KeysU.Pas as:

```
procedure SendKeys(
  const Keys: String);
```

So SendKeys expects a string, which should be made up of character representations of the virtual key codes of all the keys that need to be individually pressed and released. This is why vk_SnapShot is passed to the Chr function, to make a value of type Char, before passing it to the SendKeys routine. Also, remember that even though you may want to simulate Alt+Print Screen, you should forget about the Alt key, since Print Screen is a special case with regard to faked keystrokes. In fact the checkbox on the form helps deal with this. As you check and uncheck it, it changes the value of a Boolean typed constant in the KeysU unit called SnapShot WholeScreen, which is used if the vk_SnapShot virtual key is passed through to SendKeys:

```
SendKeys(Chr(vk_SnapShot));
ImgClipBoard.Picture.Assign(
  ClipBoard);
```

➤ Figure 2



SendKeys is fairly useful if you wish to create individual keystrokes by different keys, but sometimes (in fact often) this is not enough. In many cases you need to be able to hold one key down whilst pressing one or more other keys, and then later release the original key. For these cases, there are two other routines in the KeysU unit: PressKey and ReleaseKey, both of which take one character as a parameter.

Let's take a look at the Delphi button on the form. The job of this button is to invoke the Delphi About box and type in the Easter Egg character sequence indicated by the spin edit. For example, Alt+TEAM gives you a scrolling list of the entire Delphi development team. The idea is that Alt must be kept pressed whilst you type the letters T, E, A and M.

The way the code operates is to see if the About box is already showing. If it is, it closes it with a press of the Escape key to make sure that the rest of the code starts off in a consistent state, with no About box showing. Next, the Delphi main window (which is a form called AppBuilder, of type TAppBuilder) is brought to the foreground. This may not be enough to make it visible, however: if Delphi is minimised, the main form is in fact hidden. Regardless, though, the keystrokes necessary to invoke the About box are then manufactured (Alt, H, A). Once the About box appears, the Easter Egg key sequence is sent by pretending to hold the Alt key down (with PressKey), sending the appropriate keystrokes along (with SendKeys), and finally releasing the Alt key (with ReleaseKey).

Listing 2 shows the code. The details of the ScreenHas256ColoursOrMore function are not important, but can be found in the project on the disk. It merely calls some Windows API routines to work out if the Alt+AND Easter Egg from Delphi 1 stands any chance of working.

You might notice that I was careful to pass the character keypress values as upper cased letters. This is a requirement: the virtual key

➤ Listing 2

```
procedure TForm1.btnDelphiClick(Sender: TObject);
var
  Wnd: HWnd;
begin
  Wnd := FindWindow('TAboutBox', 'About Delphi');
  { Get rid of About box if it happens to be up so we know where we are }
  if Wnd <> 0 then begin
    BringWindowToTop(Wnd);
    SendKeys(Chr(vk_Escape));
  end;
  { Find Delphi's main window - note the Delphi }
  { caption changes, so we'll use nil for the caption }
  Wnd := FindWindow('TAppBuilder', nil);
  if Wnd = 0 then
    Exit;
  { If Delphi is minimised, this statement may have no visible effect }
  BringWindowToTop(Wnd);
  { Delphi 1 has four About box gang screens, Delphi 2 has three, }
  { Delphi 3 has three and Delphi 4 has four. The fourth one }
  { (Delphi 1 only) only works on >=256 colour screen drivers }
  SendKeys(Chr(vk_Menu)+'HA'); { Invoke the About box: Help | About }
  PressKey(Char(vk_Menu)); { Hold down Alt key }
  case edtGangScreen.Value of
    1 : SendKeys('DEVELOPERS');
    2 : SendKeys('TEAM');
    3 : SendKeys('VERSION');
    4 : if ScreenHas256ColoursOrMore then
          SendKeys('AND')
        else
          MessageDlg(
            'Alt+AND (Delphi 1 only) requires at least a 256-colour driver',
            mtInformation, [mbOk], 0);
    5: SendKeys('QUALITY');
    6: SendKeys('CHUCK');
  end;
  ReleaseKey(Char(vk_Menu));  { Release Alt key }
end;
```

codes of character keys are represented by the ordinal value of their upper case character (again refer to Issue 38, p59). In fact the only keys that have real virtual key codes defined are the non-printable characters (with the exception of just a couple of keys such as `Tab`, `Enter` and `Space`).

Things are made rather more difficult by additional printable characters which are not alphanumeric, and any requirement to enter mixed case characters. The third button on the form in Figure 2 is designed to enter a string into an edit control with one upper case letter, several lower case letters and also a colon and space. In order to find out what is required here, we can help ourselves quite a lot by making use of WinSight once again. Figure 3 shows what key up and down messages are generated by manually typing most of my desired string into the edit box.

What this tells us, rather explicitly, is that the keypresses and releases I need to get everything up to (but not including) the copyright symbol (assuming `Caps Lock` is off) are those shown in Listing 3.

As far as virtual key codes go, `Shift` and `Space` have been accurately listed as `vk_Shift` and `vk_Space`. All the letters have virtual key codes listed by WinSight, but in Delphi we use the ordinal value of the uppercase version of the letter. The one remaining sticky point is the colon (or semicolon) key. This is listed as some strange constant, `vk_FFBA` (a symbol which does not exist).

To find out which value we should be using, look at the low byte of the `WParam` value for the message in question. WinSight uses the short name `wp` for `WParam` and gives it a value of `$000000BA`. The low byte of this value is `$BA`. So `$BA` is the virtual key code for the `semicolon` key. Unfortunately there are no predefined symbols for many of these non-alphanumeric printable keys, despite my suggesting there are in Issue 38.

So after all this investigation we should now understand Listing 4, which is the `OnClick` event handler for the form's `Copyright` button. Note that the code also checks whether `Caps Lock` is on or off. If it is on (indicated by the lowest bit in the `GetKeyState` return value being set, thereby making it an odd number), it is turned back off.

The routines that do all the work in the `KeysU` unit are all based around calls to `KeyBd_Event`. In fact a procedure `PostVirtualKeyEvent` is the only thing to call `KeyBd_Event` directly. This helper routine is in turn called by `PressKey`, `ReleaseKey` and `SendKeys`. `PostVirtualKeyEvent` uses conditional compilation to cater for the differences between the 16-bit and 32-bit versions of `KeyBd_Event` (see Listing 5). You can see where the name of this helper routine came from, it is the

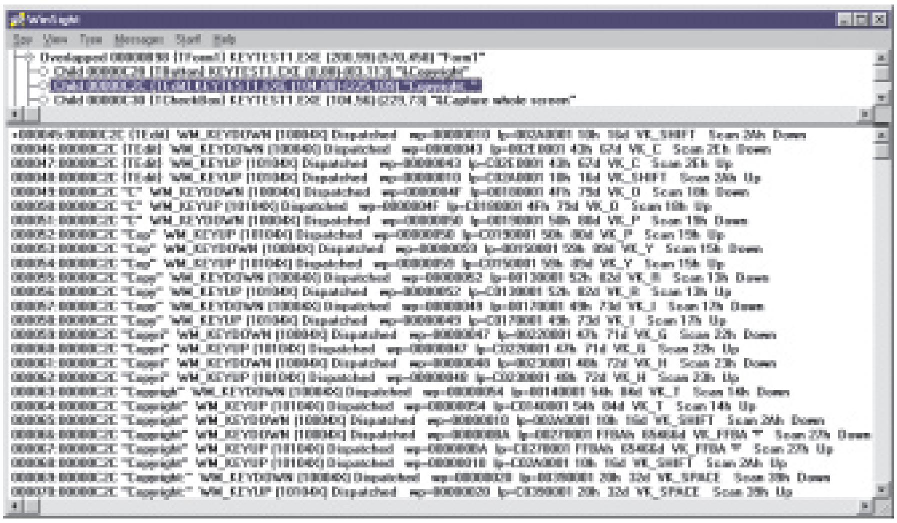
```
C: Press Shift, press C, release C, release Shift
o: Press O, release O
p: Press P, release P
y: Press Y, release Y
r: Press R, release R
i: Press I, release I
g: Press G, release G
h: Press H, release H
t: Press T, release T
Colon: Press Shift, press semicolon, release semicolon, release Shift
Space: Press Space, release Space
```

➤ *Listing 3*

```
procedure TForm1.btnCopyrightClick(Sender: TObject);
begin
  { The intention here is to enter the string: }
  {   Copyright: © }
  { into the edit control. This requires some planning }
  { to get the mixed case, as well as the colon character }
  { Give focus to edit }
  Edit1.SetFocus;
  { Make sure Caps Lock is off }
  if Odd(GetKeyState(vk_Capital)) then
    SendKeys(Chr(vk_Capital));
  { Hold down Shift key, press C, then release Shift }
  PressKey(Chr(vk_Shift));
  SendKeys('C');
  ReleaseKey(Chr(vk_Shift));
  { Press more keys (which will be lower case) }
  SendKeys('OPYRIGHT');
  { Hold down Shift key, press ;, then release Shift }
  PressKey(Chr(vk_Shift));
  SendKeys(#$BA);
  ReleaseKey(Chr(vk_Shift));
  { Send a space character }
  SendKeys(Chr(vk_Space));
  { Do Alt+0169 on number pad }
  PressKey(Chr(vk_Menu));
  SendKeys(Chr(vk_Insert) + Chr(vk_End) + Chr(vk_Right) + Chr(vk_Prior));
  ReleaseKey(Chr(vk_Menu))
end;
```

➤ *Listing 4*

➤ *Figure 3*



```
Overlapped 000008B0 {TForm1} KEYTEST1.EXE (200,99)-(570,458) "Form1"
  Child 00000C28 {TButton} KEYTEST1.EXE (8,88)-(83,113) "&Copyright"
  Child 00000C2C {TEdit} KEYTEST1.EXE (104,88)-(225,109) "Copyright: "
  Child 00000C30 {TCheckBox} KEYTEST1.EXE (104,56)-(225,73) "&Capture whole screen"
```

```
•000045:00000C2C {TEdit} WM_KEYDOWN (0100h) Dispatched wp=00000010 lp=002A0001 10h 16d VK_SHIFT Scan 2Ah Down
000046:00000C2C {TEdit} WM_KEYDOWN (0100h) Dispatched wp=00000043 lp=002E0001 43h 67d VK_C Scan 2Eh Down
000047:00000C2C {TEdit} WM_KEYUP (0101h) Dispatched wp=00000043 lp=C02E0001 43h 67d VK_C Scan 2Eh Up
000048:00000C2C {TEdit} WM_KEYUP (0101h) Dispatched wp=00000010 lp=C02A0001 10h 16d VK_SHIFT Scan 2Ah Up
000049:00000C2C "C" WM_KEYDOWN (0100h) Dispatched wp=0000004F lp=00180001 4Fh 79d VK_O Scan 18h Down
000050:00000C2C "C" WM_KEYUP (0101h) Dispatched wp=0000004F lp=C0180001 4Fh 79d VK_O Scan 18h Up
000051:00000C2C "C" WM_KEYDOWN (0100h) Dispatched wp=00000050 lp=00190001 50h 80d VK_P Scan 19h Down
000052:00000C2C "Cop" WM_KEYUP (0101h) Dispatched wp=00000050 lp=C0190001 50h 80d VK_P Scan 19h Up
000053:00000C2C "Cop" WM_KEYDOWN (0100h) Dispatched wp=00000059 lp=00150001 59h 89d VK_Y Scan 15h Down
000054:00000C2C "Cop" WM_KEYUP (0101h) Dispatched wp=00000059 lp=C0150001 59h 89d VK_Y Scan 15h Up
000055:00000C2C "Copy" WM_KEYDOWN (0100h) Dispatched wp=00000052 lp=00130001 52h 82d VK_R Scan 13h Down
000056:00000C2C "Copy" WM_KEYUP (0101h) Dispatched wp=00000052 lp=C0130001 52h 82d VK_R Scan 13h Up
000057:00000C2C "Copy" WM_KEYDOWN (0100h) Dispatched wp=00000049 lp=00170001 49h 73d VK_I Scan 17h Down
000058:00000C2C "Copy" WM_KEYUP (0101h) Dispatched wp=00000049 lp=C0170001 49h 73d VK_I Scan 17h Up
000059:00000C2C "Copyri" WM_KEYDOWN (0100h) Dispatched wp=00000047 lp=00220001 47h 71d VK_G Scan 22h Down
000060:00000C2C "Copyri" WM_KEYUP (0101h) Dispatched wp=00000047 lp=C0220001 47h 71d VK_G Scan 22h Up
000061:00000C2C "Copyri" WM_KEYDOWN (0100h) Dispatched wp=00000048 lp=00230001 48h 72d VK_H Scan 23h Down
000062:00000C2C "Copyri" WM_KEYUP (0101h) Dispatched wp=00000048 lp=C0230001 48h 72d VK_H Scan 23h Up
000063:00000C2C "Copyrigh" WM_KEYDOWN (0100h) Dispatched wp=00000054 lp=00140001 54h 84d VK_T Scan 14h Down
000064:00000C2C "Copyright" WM_KEYUP (0101h) Dispatched wp=00000054 lp=C0140001 54h 84d VK_T Scan 14h Up
000065:00000C2C "Copyright" WM_KEYDOWN (0100h) Dispatched wp=00000010 lp=002A0001 10h 16d VK_SHIFT Scan 2Ah Down
000066:00000C2C "Copyright" WM_KEYDOWN (0100h) Dispatched wp=000000BA lp=00270001 FFBAh 65466d VK_FFBA Scan 27h Down
000067:00000C2C "Copyright" WM_KEYUP (0101h) Dispatched wp=000000BA lp=C0270001 FFBAh 65466d VK_FFBA Scan 27h Up
000068:00000C2C "Copyright" WM_KEYUP (0101h) Dispatched wp=00000010 lp=C02A0001 10h 16d VK_SHIFT Scan 2Ah Up
000069:00000C2C "Copyright:" WM_KEYDOWN (0100h) Dispatched wp=00000020 lp=00390001 20h 32d VK_SPACE Scan 39h Down
000070:00000C2C "Copyright:" WM_KEYUP (0101h) Dispatched wp=00000020 lp=C0390001 20h 32d VK_SPACE Scan 39h Up
```

```
const
  SnapShotWholeScreen: Boolean = False;
  KeyEventF_KeyDown = 0;
{$ifndef WIN32}
  KeyEventF_KeyUp = $80; {It changes to 2 in Win32}
procedure KeyBd_Event; far; external 'USER' index 289;
procedure PostVirtualKeyEvent(vk: Word; fUp: Boolean);
var
  AXReg, BXReg: WordRec;
const
  ButtonUp: array[Boolean] of Byte =
    (KeyEventF_KeyDown, KeyEventF_KeyUp);
begin
  AXReg.Hi := ButtonUp[fUp];
  AXReg.Lo := vk;
  BXReg.Hi := 0; { not an extended scan code }
  { Special processing for the Print Screen key. }
  { If scan code is set to 1 it copies entire }
  { screen. If set to 0 it copies active window. }
  if vk = vk_SnapShot then
    BXReg.Lo := Byte(SnapShotWholeScreen)
  else
    BXReg.Lo := MapVirtualKey(vk, 0);
  asm
    mov ax, AXReg
    mov bx, BXReg
    call KeyBd_Event
  end;
end;
{$else}
procedure PostVirtualKeyEvent(vk: Word; fUp: Boolean);
var
  ScanCode: Byte;
const
  ButtonUp: array[Boolean] of Byte =
    (KeyEventF_KeyDown, KeyEventF_KeyUp);
begin
  if vk = vk_SnapShot then
    { Special processing for the Print Screen key. }
    { If scan code is set to 1 it copies entire }
    { screen. If set to 0 it copies active window. }
    ScanCode := Byte(SnapShotWholeScreen)
  else
    ScanCode := MapVirtualKey(vk, 0);
  KeyBd_Event(vk, ScanCode, ButtonUp[fUp], 0);
end;
{$endif}
procedure PressKey(Key: Char);
begin
  PostVirtualKeyEvent(Ord(Key), False)
end;
procedure ReleaseKey(Key: Char);
begin
  PostVirtualKeyEvent(Ord(Key), True)
end;
procedure SendKeys(const Keys: String);
var
  Loop: Byte;
begin
  for Loop := 1 to Length(Keys) do Begin
    { Press key }
    PostVirtualKeyEvent(Ord(Keys[Loop]), False);
    { Release key }
    PostVirtualKeyEvent(Ord(Keys[Loop]), True);
  end;
  { Let the keys be processed }
  Application.ProcessMessages;
end;
```

➤ *Listing 5*

same as the old 16-bit Pen Windows keystroke manufacturing API name referred to earlier.

A second version of the project, KeyTest2.Dpr is also supplied. This works exactly the same as the first version of the project with one difference. Instead of manufacturing keystrokes using `KeyBd_Event`, it uses the new `SendInput` API. The implications of this are that the program will only run on Windows 98 or later, or on Windows NT 4.0 with Service Pack 3 or later. Also, it will only compile in Delphi 4 or later. In all other respects it matches the first project. The implementation of `PostVirtualKeyEvent` now looks like Listing 6.

```
procedure PostVirtualKeyEvent(vk: Word; fUp: Boolean);
var
  ScanCode: Byte;
  Input: TInput;
const
  KeyEventF_KeyDown = 0;
  //This constant is defined incorrectly in Delphi 4
  Input_KeyBoard  = 1;
  ButtonUp: array[Boolean] of Byte = (KeyEventF_KeyDown, KeyEventF_KeyUp);
begin
  if vk = vk_SnapShot then
    { Special processing for the PrintScreen key. }
    { If scan code is set to 1 it copies entire }
    { screen. If set to 0 it copies active window. }
    ScanCode := Byte(SnapShotWholeScreen)
  else
    ScanCode := MapVirtualKey(vk, 0);
  FillChar(Input, SizeOf(Input), 0);
  Input.IType := Input_KeyBoard;
  Input.KI.wVk := vk;
  Input.KI.wScan := ScanCode;
  Input.KI.dwFlags := ButtonUp[fUp];
  Input.KI.time := GetTickCount;
  SendInput(1, Input, SizeOf(Input))
end;
```

➤ *Listing 6*

A couple of noteworthy points arise about the Delphi 4 support for `SendInput`. In Delphi 4, there are two mistakes in the Delphi translations of the types and constants used by `SendInput`. Firstly, as you can see from Listing 6, the constant `Input_KeyBoard` (and also `Input_Hardware`) are defined incorrectly as zero. Instead, `Input_KeyBoard` should be 1 and `Input_Hardware` should be 2. Also, the `TInput` record has three variant parts: a `TMouseInput` record, a `TKeyBdInput` record and a `THardwareInput` record. Each of these records has a field for passing extra information (as in the fourth parameter to `KeyBd_Event`). However, these are all incorrectly named `dwExtractInfo` instead of `dwExtraInfo`. The constants are fixed in the second update to Delphi 4, but not the record field names.

Brian Long is an independent consultant and trainer. You can reach him at brian@blong.com

Copyright @ 1998 Brian Long
All rights reserved.