Trapping Keystrokes

by Warren Kovach

Typing accented characters in Windows programs can be problematic, particularly if you have an English language keyboard. Keyboards designed for non-English languages will have keys for some accented characters, or allow them to be entered with two or three keystrokes. However, they commonly have just those few accented characters that are most useful for that language. The US-International keyboard layout also allows accented characters to be entered with several keystrokes, but this assumes the US keyboard layout, which is different to the UK keyboard I use. Some word processors, such as Word, have their own set of shortcuts for accented characters, but these only work within that program and are different to the usual combinations on non-English keyboards.

In the absence of any of these schemes more laborious techniques must be used to enter accented and other extended characters. You can either cut and paste from the Windows Character Map app or type the character's code on the numeric keypad (eg Alt-0228 for ä). Hands up everyone who has a chart of these codes taped up next to your computer!

What is needed is a consistent and easy to remember scheme for producing all accented and extended characters that will work on any keyboard.

Several years ago, soon after I started using Windows, I ran across a freeware program called Compose, produced by Digital Equipment Corporation to emulate the Compose key found on many of their own terminals and workstations. When this key was pressed and released the typist could then press two other keys to compose an accented character. The sequence `Compose`, `a` and `"` would produce ä. I regularly need to use accented characters in many different Windows programs, so this application quickly became a permanent part of my setup.

When Windows 95 was released I discovered to my dismay that the 16-bit Compose program would not work with 32-bit programs. I resolved to write my own 32-bit version. Someday.

That day finally arrived when I started looking at ways to trap system-wide keystrokes. Trapping keystrokes within your own application is easy: create an event handler for `OnKeyPress`, `OnKeyDown` or `OnKeyUp` for the appropriate form or control. If you want the form to get the keystrokes for all its controls you set `KeyPreview` to `True`.

Trapping keystrokes from other programs is more difficult: you have to drop down to the Windows API level and implement something called a hook. This article shows you how to go about installing a system-wide keyboard hook to allow entry of accented characters. Since it is inspired by the Compose program and is written in Delphi, I've called the program DCompose (no jokes about smelly things, please!).

Windows Hooks

A hook is a mechanism for intercepting various system events before they reach an application. These events can be mouse movements, keystrokes or various other Windows messages. The hook consists of a procedure that, once registered with Windows, will receive notification of the desired type of events. That procedure can then examine the event and act on it.

Windows maintains a chain of hook procedures so that more than one program can intercept the events. Once a hook procedure has examined and acted on an event it will usually pass it on to the next hook in the chain. Of course there will be situations where the event will not be passed on. For example, the purpose of a hook may be to disable a certain type of keystroke. In that case the procedure will examine each keystroke and only pass on those that are not disabled. A hook procedure such as this must be written carefully to ensure that all other events are indeed passed on. A rogue hook procedure that swallows all keystrokes could have a very detrimental effect on other programs!

There are three API functions used for implementing hooks: `SetWindowsHookEx`, `UnhookWindowsHookEx` and `CallNextHookEx`. The first two take care of registering the hook procedure with Windows and removing it from the chain when finished. `CallNextHookEx` is used to pass an event on to the next hook in the chain.

The `SetWindowsHookEx` function is declared as follows:

```
function SetWindowsHookEx(
  idHook: Integer;
  lpfn: TFNHookProc;
  hmod: HINST;
  dwThreadId: DWORD): HHOOK;
  stdcall;
```

The `HINST`, `DWORD` and `HHOOK` types are `Integer` in 32-bit Delphi. This function takes four parameters. The first, `idHook`, is the type of hook being installed. There are 12 types, ranging from `WH_CALLWNDPROC`, which traps all messages sent with `SendMessage`, to `WH_CBT` for examining a variety of window and input events to implement a computer based training system, and `WH_MOUSE` to intercept mouse messages. We will use the `WH_KEYBOARD` hook, which traps all keystrokes.

The second parameter, `lpfn`, is a pointer to the hook procedure. This is the procedure that you write to examine and act on the events. It must be declared with the following parameters:

```
function MyHookProcedure(
  Code: Integer;
  wParam: WPARAM;
  lParam: LPARAM) : LRESULT;
```

The meaning of the three parameters varies depending on the type of hook that was installed. However, in general the Code parameter gives information about the action that caused the event to be triggered, or how the procedure should deal with it. wParam and lParam, which are both 32-bit integers in Windows 95 and NT, give information about the event itself. For the WH_KEYBOARD hook wParam will contain the virtual-key code for the keystroke. This is a device-independant code that specifies the key that has been pressed. For example, the VK_A code specifies the key that produces the letter 'A', which on French and Belgian keyboards is in a different position to most others. lParam contains extra information on the keystroke, such as whether the key is being pressed or released, if the Alt key is being pressed, and the number of times the key has been repeated if the user is holding it down.

Your hook procedure must return a 32-bit integer result. The result to return again varies depending on the type of hook. For WH_KEYBOARD a non-zero result means that Windows should not pass the keystroke on to the application or any other hook procedure. A zero result means that the keystroke can be sent to the appropriate application.

The final two parameters for SetWindowsHookEx, hMod and dwThreadID, let you specify either the DLL or thread to which the hook applies. If you want a system-wide hook you pass the instance handle of the DLL containing the hook procedure in hMod. If instead you want the hook to only work within a particular thread then you pass the thread ID in dwThreadID. When one of these parameters is set the other should be 0.

When you are done using a hook you must remove the procedure from the chain. This is done with the folowing function:

```
function UnhookWindowsHookEx(
  hhk: HHOOK): BOOL;
```

The hhk parameter is the hook handle that was returned by SetWindowsHookEx, so you must make sure to save this when you set up the hook.

The hook handle is also needed when passing an event on to the next hook procedure in the chain, which is done with the following function:

```
function CallNextHookEx(
  hhk: HHOOK;
  nCode: Integer;
  wParam: WPARAM;
  lParam: LPARAM): LRESULT;
```

Besides the current hook handle hhk (which is ignored by current versions of Windows, but may be used in the future), your hook procedure must also pass on the three parameters it received. The result returned by CallNext- HookEx will have the same meaning as that of your own hook procedure.

Your hook procedure would therefore normally take the result from CallNextHookEx and return it as its own result. In this way the result code filters up the chain of hook procedures back to Windows.

Designing The Hook

Now that we know the basics of how to set up a hook how can we use this to solve our problem? We need to trap a sequence of three keystrokes and convert them into a single accented character. All other keystrokes must be passed on unchanged.

The primary task of our hook procedure will be to examine all keystrokes, looking for our 'Compose' key being pressed. I've followed the lead of Digital and specified that the right-hand Ctrl key will be the Compose key. You could just as easily use some other key, such as F12, but you must be careful that you are not taking over a key that is vital to some other program. Since there are two Ctrl keys, hijacking one of them for our own purposes is least likely to cause disruptions. A more sophisticated program should give the user a choice of keys to use.

Once the Compose key has been pressed the program must go into compose mode. The first thing it should do is to visually alert the user that compose mode is now active. In Windows 95 and NT 4 the easiest way to do this is to have an icon in the TaskBar tray (or the notification area) that changes to indicate the current mode. The details of how to do this are given later in this article.

In compose mode our program must trap the next two keystrokes. It will then compare these to a list of possible key combinations. If they match one of these then the appropriate accented character (or other extended character) is then inserted into whichever program was previously receiving text. If no match is found then the two characters are inserted separately. A beeping noise is generated to warn the user of an unexpected event.

System-wide hooks must usually be implemented in a DLL (the exceptions are the WH_JOURNALRECORD and WH_JOURNALPLAYBACK hooks used for recording and playing back sequences of messages). This is required so that the DLL (and its hook procedure) can be mapped into the address space of whatever program is triggering keystroke events. If the hook procedure were in an application not a DLL it would only be able to trap keystrokes for that application, not any others.

This gives us the chance to divide the tasks described above, putting some in the DLL and some in the parent program. I've decided to make the DLL as simple as possible. It takes care of trapping and examining the keystrokes. If a compose sequence has been typed then it notifies the parent program, passing it the sequence typed and the handle of the window that was receiving the keystrokes. The parent program then does the translation and enters the accented character into appropriate window. This opens the possibility of allowing the user to customize the list of key sequences; it would be much more difficult to do this if the translation occurred in the DLL.

User-Defined Messages

To work together the DLL and parent program must talk to each other. This is best done with user-defined Windows messages. A Windows message such as WM_KEYDOWN is simply a constant that tells the Windows system and its applications how to interpret the data in the lParam and wParam parameter. Windows reserves the numbers 0 to 1023 for its own messages. Any numbers above this can be used by applications for their own private messages.

All you need to do is declare a constant with a number above this. The constant WM_USER is 1024, so your messages can use this to ensure that they don't encroach on Windows message space. As an example, you can create some messages like this:

```
const
  WM_MyFirstMessage = WM_USER+100;
  WM_MySecondMessage = WM_USER+101;
```

This is fine if the messages are only ever used within your own appli--cation. However, our key-trapping DLL communicates with the parent application by broadcasting a user-defined message to all top-level windows, rather than directly to the parent app's window (the reasons will become clear later).

Now what happens if our DLL broadcasts a WM_MyFirstMessage and another application is running that has also defined a private message as WM_USER+100? Yep, the function in that other application that is linked to that message will be triggered, with potentially disastrous consequences. The golden rule when sending user-defined messages between modules or applications is that the message constant *must* be unique.

Fortunately Windows helps us out with this by providing the RegisterWindowMessage API function. This takes a string parameter that describes the message and returns a unique message constant. If another application calls RegisterWindowMessage with an identical string parameter then the same constant is returned. So, if we put the following line of code in all applications and DLLs that must use a private message we are guaranteed that our message won't affect any other applications (unless they happen to have used an identical string!):

```
WM_MyFirstMessage :=
  RegisterWindowMessage(
  'My unique WM_MyFirstMessage');
```

We will use this technique to register two messages to allow the DLL to communicate with the parent application. WM_ToggleIcon will tell the parent program to change the TaskBar icon to indicate it is in compose mode. WM_TranslateKeys will send the resulting keystroke sequence for translation.

Broadcasting Messages

In the hook procedure described below messages are sent to the parent application by broadcasting them to all top-level windows, using the HWND_BROADCAST identifier rather than a specific window handle. You might think it would be easier to use the parent app's handle here. You could, for example, pass the handle to the DLL as a parameter to the EnableHook initialization function, then store it in a global variable. However, this will only work when the parent app's main form has the focus; if compose sequences are typed in other apps nothing happens.

The reason for this is that, under 32-bit Windows, a separate instance of a DLL is mapped into the address space of each client process. In this case, where the DLL contains a Windows hook, each process running on the computer will be a client. Each of these instances will have its own set of global variables. The one that corresponds to the DCompose application will have the ParentHandle global variable correctly set but the others will be set to zero. The keystrokes will be trapped properly in the other apps but the zero value will be passed to PostMessage, where it will be interpreted as *post this message to the current thread*; this is not the result we want.

There are methods for allowing multiple instances of DLLs to share common memory. These include using Windows memory mapped files or the Delphi Shared Memory Manager in the ShareMem unit. However, these are over-complex.

On the positive side, having separate sets of variables for each instance can be an advantage. You needn't worry about a compose sequence meant for one window winding up in another. Once you've got DCompose running try typing the first letter of a compose sequence in one application, then switch to another and type another sequence. When you go back to the first program and finish the sequence the correct character will be entered.

Installing The Hook

Now we are ready to implement the DLL. The full source code is shown in Listing 1. Note that there are several compiler directives that allow it to work under 16-bit Delphi 1 as well (see later).

First, a warning. All keystroke messages for the entire system will pass through this hook procedure. To avoid degrading system performance you should make sure that as little processing takes place here as possible.

The uses clause contains only Windows and Messages. When creating a new DLL project Delphi will automatically put in SysUtils and Classes. We can take these out and reduce the DLL to just 16Kb.

There are several global variables. Hook stores the handle of the current hook, which is needed for the calls to CallNextHookEx and UnhookWindowsHookEx. Next are the two private messages used to communicate with the parent application. Two boolean variables, ComposeMode and FirstKeyEntered, keep track of where we are in a compose sequence. The ASCII values of the two keys entered are stored in FirstKey and SecondKey.

The calling program initializes the hook procedure by calling the EnableHook procedure, which initializes the two global boolean variables then calls SetWindowsHookEx to register the hook procedure. By passing it the handle of the DLL (hInstance) we are telling Windows this is a system-wide hook. The hook is removed by calling the DisableHook procedure.

```
library KeyHook;
uses
{$IFDEF Ver80}
  WinProcs,
  WinTypes;
{$ELSE}
  Windows,
  Messages;
{$ENDIF}
var
  Hook : HHOOK;
  WM_ToggleIcon,
  WM_TranslateKeys : Cardinal;
  ComposeMode, FirstKeyEntered : boolean;
  FirstKey,SecondKey : word;
{$IFDEF Ver80}
type
  wParam = Word;
  lParam = Longint;
  LResult = Longint;
{$ENDIF}

function HookProcedure(nCode: Integer; vkCode: WPARAM;
  MsgInfo: LPARAM) : LRESULT;
  {$IFDEF Ver80} export {$ELSE} stdcall {$ENDIF};
const
  ExcludedKeys = [VK_SHIFT, VK_MENU, VK_CONTROL, VK_LWIN,
    VK_RWIN, VK_APPS];
var
  KeyState : TKeyboardState;
  Buffer : array[0..2] of char;
  ToASCIIResult : integer;
begin
  if (nCode = HC_ACTION) then begin
    { by default, don't allow keystroke to be passed to
      designated window; this will be changed if our three
      Compose checks fail and we call CallNextHookEx }
    Result := 1;
    { check if right-hand control key is being released }
    if (vkCode = VK_CONTROL) and
      ((HiWord(MsgInfo) and KF_UP) <> 0) and
      ((HiWord(MsgInfo) and KF_EXTENDED) <> 0) then begin
      PostMessage(HWND_BROADCAST,WM_ToggleIcon,1,0);
      ComposeMode := true;
      { bypass CallNextHookEx, since we don't want
        right control key going to app }
      exit;
    end else if (ComposeMode) and
      ((HiWord(MsgInfo) and KF_UP) = 0) and
      { key being pressed } (not (vkCode in ExcludedKeys))
      then begin
      { key not the shift key }
      if not FirstKeyEntered then begin
        GetKeyboardState(KeyState);
       {$IFDEF Ver80}
        ToASCIIResult := ToASCII(vkCode,
          MapVirtualKey(vkCode, 0), @KeyState, @Buffer, 0);
       {$ELSE}
        ToASCIIResult := ToASCII(vkCode,
          MapVirtualKey(vkCode, 0), KeyState, Buffer, 0);
       {$ENDIF}
        if ToASCIIResult = 0 then
          FirstKey := 0
        else if ToASCIIResult > 0 then
          FirstKey := ord(Buffer[0])
        else if ToASCIIResult < 0 then begin
          { a dead key has been entered; exit compose
            mode and let Windows take care of composing
            character itself }
          PostMessage(HWND_BROADCAST,WM_ToggleIcon,0,0);
          ComposeMode := false;
          FirstKeyEntered := false;
          Result := CallNextHookEx(0,nCode,vkCode,MsgInfo);
           exit;
        end;
        FirstKeyEntered := true;
        exit; { bypass CallNextHook }
      end else begin
        GetKeyboardState(KeyState);
       {$IFDEF Ver80}
        ToASCIIResult := ToASCII(vkCode,
          MapVirtualKey(vkCode, 0), @KeyState, @Buffer, 0);
       {$ELSE}
        ToASCIIResult := ToASCII(vkCode,
          MapVirtualKey(vkCode, 0), KeyState, Buffer, 0);
       {$ENDIF}
        if ToASCIIResult = 0 then
          SecondKey := 0
        else if ToASCIIResult > 0 then
          SecondKey := ord(Buffer[0])
        else if ToASCIIResult < 0 then begin
          { a dead key has been entered; exit compose
            mode and let Windows take care of composing
            character itself }
          PostMessage(HWND_BROADCAST,WM_ToggleIcon,0,0);
          ComposeMode := false;
          FirstKeyEntered := false;
          Result := CallNextHookEx(0,nCode,vkCode,MsgInfo);
          exit;
        end;
        PostMessage(HWND_BROADCAST,WM_ToggleIcon,0,0);
        ComposeMode := false;
        FirstKeyEntered := false;
       {$IFDEF Ver80}
        PostMessage(HWND_BROADCAST, WM_TranslateKeys,
          word(FirstKey + (SecondKey shl 8)), GetFocus);
       {$ELSE}
        PostMessage(HWND_BROADCAST, WM_TranslateKeys,
          MAKELONG(FirstKey, SecondKey), GetFocus);
       {$ENDIF}
        exit; { bypass CallNextHook }
      end;
    end;
  end;
Result := CallNextHookEx(0, nCode, vkCode, MsgInfo);
end;

procedure EnableHook;
  {$IFDEF Ver80} export {$ELSE} stdcall {$ENDIF};
begin
  ComposeMode := false;
  FirstKeyEntered := false;
 {$IFDEF Ver80}
  Hook := SetWindowsHookEx(WH_KEYBOARD, HookProcedure,
    hInstance, 0);
 {$ELSE}
  Hook := SetWindowsHookEx(WH_KEYBOARD, @HookProcedure,
    hInstance, 0);
 {$ENDIF}
end;

procedure DisableHook;
  {$IFDEF Ver80} export {$ELSE} stdcall {$ENDIF};
begin
  UnhookWindowsHookEx(Hook);
end;

exports
  EnableHook, DisableHook;

begin
  WM_ToggleIcon :=
    RegisterWindowMessage('DCompose ToggleIcon');
  WM_TranslateKeys :=
    RegisterWindowMessage('DCompose TranslateKeys');
end.
```

➤ *Listing 1*

The heart of the DLL is the `HookProcedure` function. Look first at the last line where there's a call to `CallNextHookEx`. This passes the event trapped by this hook onto the next one in the chain, setting the result of our procedure to that of the next one. There are a few well-defined places in our hook procedure where we don't want to pass the event on to the next hook procedure, so we call `Exit` to leave immediately. All other events continue through to the end of the procedure and on to the next hook.

Our first action in the hook procedure is to check the `nCode` parameter. A value of `HC_ACTION` means that a keystroke event is being processed in the normal way, so we examine it and decide what to do. A value of `HC_NOREMOVE` means that an application has used `PeekMessage` to examine the message queue but has not removed the message from the queue. In this case we just ignore the event and go directly to the end of the function where it is passed to `CallNextHookEx`. In some circumstances `nCode` can be a negative value. Early versions of Windows used these to maintain the hook chain, but they are no longer needed. However, it is recommended that a hook procedure ensure that `CallNextHookEx` is called if a negative value occurs.

Examining Keystrokes

Once we've received notification of a keyboard event we can examine it. First we set the function `Result` to a non-zero value. This tells Windows not to pass the keystroke on to the current application, which is what we want if the keystroke is part of a compose sequence. If it is not then `Result` will be reset to the return value of the CallNextHookEx function.

We initially want to trap our Compose key, which is the right-hand Ctrl key. We need to look at three things to determine if that key has been pressed. First we check whether the wParam parameter (here given a more understandable name of vkCode) is equal to VK_CONTROL. This value is returned by both control keys. To distinguish left from right we check if the KF_EXTENDED flag is set in the MsgInfo parameter. The extended keys on the standard PC keyboard are all those to the right of the main block of alphanumeric keys, such as the numeric keypad and navigation keys. They also include the right hand Ctrl and Alt keys. Finally we check the KF_UP flag to see if the key is being released.

If these three conditions are met the Compose key has been pressed, so we go into compose mode by setting the ComposeMode variable to True and sending a message to the parent application, telling it to provide a visual clue that we are now waiting for a compose sequence. We then exit the function immediately, bypassing the CallNextHookEx call.

The next time a keystroke comes through this function ComposeMode will be True, so we can now examine and retrieve the two characters of the compose sequence. Again we check the KF_UP flag so that we only process key releases. We also ignore any presses of the Shift key; this is necessary to allow upper case accented letters to be composed and for the symbols above the numeral keys to be used. Several other keys, such as Ctrl and Alt, are also excluded.

The first time through in compose mode the FirstKeyEntered variable will be False, so we place the ASCII value of the first key of the sequence into FirstKey, then set FirstKeyEntered to True. The second time through the ASCII value is placed in SecondKey.

The virtual key codes passed to our hook procedure simply indicate which key on the keyboard was pressed; 'a' and 'A' will return the same code. To distinguish the different characters we must convert them to ASCII. This is done with two Windows API functions, GetKeyboardState and ToASCII. The first gets the state (up, down or toggled) of each of the virtual keys and stores them in an array. This array is then passed as the third parameter to the ToASCII function, where it is examined to determine the state of the Shift and Caps Lock keys.

ToASCII also takes several other parameters. First is the virtual key code of the key, which has been passed to the hook procedure in vkCode. The function must also be passed the hardware scan code of the relevant key. This is different to the virtual key code and the same virtual key can have different hardware scan codes on different types of keyboards, depending on the location of the key. Fortunately Windows provides the MapVirtualKey function to translate between the two. The last parameter for ToASCII indicates whether a menu is active.

The fourth parameter of ToASCII is a buffer to hold the ASCII translation. Usually this will be a single character, but in some circumstances it could be two characters. Some keyboards will have accent or diacritic keys that modify the next character typed. For example, some French keyboards have a circumflex key (^). When this is pressed nothing appears on the screen, but if the next key pressed is an 'o', the resulting character is ô. Keys that have no action themselves but modify the next key are called dead keys.

If the next key after the dead key is not a character that can be combined with a circumflex then ToASCII places both characters (^ and o) in the buffer. If the return value of ToASCII is greater than zero it indicates how many characters are in the buffer. In most circumstances it will be one character, so we place that in the appropriate variable (FirstKey or SecondKey).

The only time there will be two characters is when a dead key has been pressed. However, when this happens it means the current keyboard layout itself supports producing accented characters. Thus if we detect a dead key we should abandon our attempts to produce an accented character and let Windows get on with it itself. ToASCII will return a negative value when a dead key is entered; when this happens we turn off the compose mode and pass the key on to the next hook or the application.

Once we have successfully trapped the two keys of the compose sequence we leave compose mode by resetting the boolean variables to false and telling the parent program to switch the icon back to its default state. We then use the WM_TranslateKeys message to pass the sequence on to the parent program for translation to an accented character. These are both passed in the wParam parameter, after the Windows MAKELONG function is used to combine them. The lParam parameter is used to pass the handle of the window where the keys were typed. This is retrieved using the API function GetFocus. The DLL's job is done.

The Parent Application

The main DCompose program takes care of actually translating the compose sequence and inserting the resulting character into the appropriate text window. It also performs some housekeeping. The first thing it must do is set up the user interface.

This program takes advantage of the Windows 95/NT4 TaskBar Tray (also called the notification or status area). It places an icon on the Tray that changes colour when compose mode is active. Right clicking on the icon will bring up a menu allowing you to change options or exit. The icon is the red letter Á in Figure 1. Details of how to produce Tray icon programs were covered by Marco Cantù in the August 1996 issue, so I won't go into detail here. The code for initializing and removing the tray icon is in the FormCreate and FormDestroy methods in Listing 2.

Initializing the Tray icon is simple. First you need to load an icon from a resource file with LoadIcon; we'll store it in the form's own Icon property. Remember to add the icons to a new resource file (called DComp.res here) instead of the project's resource file (DCompose.res), otherwise your icons will be overwritten when you next compile!

Next we initialize a TNotifyIconData structure with the information the Tray needs; this is then sent to the system tray with the Shell_NotifyIcon API function. Of particular interest is the uCallBackMessage field, which points to a user-defined message we will use to communicate with the Tray icon. This is linked to the IconTray method of the form. That method simply displays a popup menu at the cursor position if the right mouse button is clicked over the icon. If the left button is double clicked on the icon the Options1Click method is invoked; this simply shows the main form, which is a modal dialog box.

One trick Marco showed in his article was how to ensure that the program doesn't create a button on the TaskBar, in addition to the icon in the Tray. This is done by disabling the creation of the hidden application window. To do this set the global system variable IsLibrary to True in the initialization section of the RunFirst unit:

```
unit RunFirst;
interface
implementation
initialization
  IsLibrary := True;
end.
```

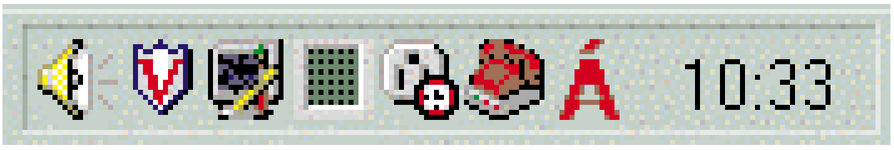
This unit is placed first in the uses clause of the program source (as in Listing 3) to ensure it is called before an attempt is made to create the application window. IsLibrary is set back to False at the start of the program source. This trick was necessary in Delphi 2, but Delphi 3 takes care of hiding the TaskBar button itself if Application.ShowMainForm is set to False. We use compiler directives to ensure this method is only used with Delphi 2.

Listing 3 shows another simple trick. We need to ensure that only one copy of DCompose is running at any one time. There are a variety of ways to do this. Several are described in my book *Delphi 3 User Interface Design*, and two others were covered in the December 1996 issue (pages 26 and 54). The method used above is to create a Windows mutex. Mutexes are objects that allow 32-bit Windows programs to gain mutually exclusive access to program resources. To use them we call the CreateMutex function, passing a string unique to our project. If the mutex already exists the ShowLastError function will return ERROR_ALREADY_EXISTS; we exit the program if this happens. If the mutex doesn't exist then we continue initializing the application as normal.

The Translation Table

Next in the order of business is to set up the data structure used for translating the compose sequences to accented and extended characters. This needs to store the two characters of the sequence as well as the single resulting character. Any number of methods could be used for this, ranging from a simple array of strings to a full blown searchable data structure

➤ *Figure 1: DCompose icon in the Windows NT4 Tray*



➤ *Listing 2*

```
const
  wm_IconMessage = wm_User;
type
  TDCompForm = class(TForm)
    { ... }
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Options1Click(Sender: TObject);
    procedure About1Click(Sender: TObject);
    procedure Close1Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure FontListChange(Sender: TObject);
  private
    NotifyIcon: TNotifyIconData;
    CharList : TStringList;
    procedure AppOnMessage(var Msg: TMsg;
      var Handled: Boolean);
    procedure TranslateKeys(First,Second : word;
      Wnd:THandle);
    procedure InitDefaultList;
  public
    procedure IconTray(var Msg: TMessage);
      message wm_IconMessage;
  end;
{$R *.DFM}
{$R Dcomp.res}
procedure TDCompForm.FormCreate(Sender: TObject);
begin
  { load the initial icon }
  Icon.Handle := LoadIcon(HInstance, 'LETTERRED');
  { fill the NotifyIcon data structure }
  with NotifyIcon do begin
    cbSize := sizeof(NotifyIcon);
    wnd := Handle;
    uID := 1; { icon ID }
    uCallBackMessage := wm_IconMessage;
    hIcon := Icon.Handle;
    szTip := 'DCompose';
    uFlags := nif_Message or nif_Icon or nif_Tip;
  end;
  Shell_NotifyIcon(NIM_ADD, @NotifyIcon);
  CharList := TStringList.Create;
  with CharList do begin
    { sort list alphabetically }
    Sorted := true;
    { must set this to accept duplicates, otherwise it seems
      to see the lower and upper case strings as duplicates
      and eliminates lower case }
    Duplicates := dupAccept;
  end;
  InitDefaultList;
  Application.OnMessage := AppOnMessage;
  EnableHook;
end;
procedure TDCompForm.FormDestroy(Sender: TObject);
begin
  NotifyIcon.uFlags := 0;
  Shell_NotifyIcon(NIM_DELETE, @NotifyIcon);
  CharList.Free;
  DisableHook;
end;
procedure TDCompForm.IconTray(var Msg: TMessage);
var
  Pt: TPoint;
begin
  if Msg.lParam = wm_rbuttondown then begin
    GetCursorPos(Pt);
    SetForegroundWindow(Handle);
    PopupMenu1.Popup(Pt.x, Pt.y);
  end;
  if Msg.lParam = wm_lbuttondblclk then
    Options1Click(self);
end;
procedure TDCompForm.Options1Click(Sender: TObject);
begin
  ShowModal;
end;
```

```
program dcompose;
uses
 {$IFDEF Ver90} RunFirst in 'RunFirst.pas', {$ENDIF}
  Forms, Windows,
  DCompfrm in 'DCompfrm.PAS'; {DCompForm}
{$R *.RES}
const MutexName = 'Run only one DCompose';
var   MyMutex : THandle;
begin
 {$IFDEF Ver90}
  IsLibrary := False;
 {$ENDIF}
  MyMutex := CreateMutex(nil,true,MutexName);
  if (MyMutex = 0) or (GetLastError = ERROR_ALREADY_EXISTS) then Halt;
  Application.ShowMainForm := False;
  Application.CreateForm(TDCompForm, DCompForm);
  Application.Run;
  CloseHandle(MyMutex);
end.
```

➤ *Listing 3*

like a B-tree or hash table. The latter methods are a bit over the top for a list of 100-200 items so we will opt for a simpler solution.

My first approach was to use a typed constant array of records, simply because I thought it was an interesting but rarely used method to program. Listing 4 shows the declarations for the types and the constant array for this method.

This method works fine, in that you can scan through the array comparing the compose sequence to the characters in the `Pair` field of each record, then return the contents of the `Acc` field when a match is found. However, this method isn't very flexible: the user can't change any of the sequences or add new ones. It would be nicer to allow the user to customize the list to his or her own liking.

I finally decided to store the sequences and resulting character in a `TStringList`. I chose this for several reasons. First, the string list has functions for adding and deleting items and sorting the list. This made maintenance of the list easy. Second, the `SaveToFile` and `LoadFromFile` methods provide a simple way to save the user's changes and reinstate them next

```
const
  Maxpairs = 105;
type
  TSeqPair = record
    Pair : string[2];
    Acc  : char;
  end;
  TSeqArray =
    array[1..MaxPairs] of TSeqPair;
const
  SeqTable : TSeqArray = (
    (Pair:'ss';Acc:'ß'),
    (Pair:'a`';Acc:'à'),
    (Pair:'a''';Acc:'á'),
    (Pair:'a^';Acc:'â'),
    (Pair:'a~';Acc:'ã'),
    (Pair:'a"';Acc:'ä'),
    (Pair:'a*';Acc:'å'),
    (Pair:'ae';Acc:'æ'),
    (Pair:'c,';Acc:'ç'),
    (Pair:'e`';Acc:'è'),
    { ... }
    (Pair:'Y=';Acc:'¥'),
    (Pair:'So';Acc:'§'),
    (Pair:'CO';Acc:'©');
```

➤ *Listing 4*

time the program is run. Finally, the list can be displayed in a control such as a list box by assigning the string list to the `Items` property:

```
MyListBox.Items := MyStringList;
```

Each item of the string list contains the accented or extended character, followed by the two characters of the sequence. A tab character is placed before the sequence characters so that when displayed in the list box they appear in two columns.

Of course, a list box doesn’t actually allow the user to modify the entries, so DCompose as presented here does not allow customization. This could be achieved with something like a string grid, or perhaps some edit boxes and `Modify` and `Add` buttons next to the list box, letting the user type new entries separately.

Listing 5 shows part of the `InitDefaultList` method that enters the characters into the string list. The string list is first created in the `FormCreate` method (Listing 2). There the `Sorted` property is set to `True`. The `Duplicates` property is also set to `dupAccept` here. When checking for duplicates the `TStringList` ignores case differences, so we must accept duplicates to allow both lower and upper case compose sequences.

Receiving DLL Messages

Once the Tray icon and string list are initialized we can start the keyboard hook. This is done through the `EnableHook` procedure exported from the DLL. It and its counterpart, `DisableHook`, are imported into our program statically with the following declarations:

```
implementation
procedure EnableHook; stdcall;
  external ‘KeyHook.dll’;
procedure DisableHook; stdcall;
  external ‘KeyHook.dll’;
```

We must also remember to set up the two user-defined messages, `WM_ToggleIcon` and `WM_TranslateKeys`, with calls to `RegisterWindowMessage` identical to those in the DLL; this is done in the `initialization` section of the main form’s unit. Finally we need to initialize a mechanism to trap these messages. This is done in `FormCreate` by pointing the `Application.OnMessage` event to our event handling method, `AppOnMessage`. All messages destined for the application will come through here, so we can filter out those from the DLL. With all this done the program sits back and waits for a compose sequence.

Listing 6 gives the code for the application’s message trapping routine. As mentioned earlier, messages from the DLL are broadcast to all windows that the operating system views as top-level (ie those that are not child windows). The primary top-level window of DCompose will differ in Delphi 2 and 3. Normally the hidden application window will be the top-level window. However, under Delphi 2 DCompose uses the `IsLibrary` trick described above to suppress the creation of the application window and thus its TaskBar button. In this case the window for the program’s main form will be the top level window.

The hidden application or main form windows are not necessarily the only top-level windows in a Delphi programs. If your program uses pop-up menus (as does DCompose), timers, the clipboard or drag-and-drop operations then Delphi will allocate more top-level windows to simplify message processing. We must make sure our application message handler does not trap messages sent to these as well, otherwise messages sent from the DLL will be received more than once. So, before proceeding with processing the DLL messages we compare the destination window of the message (`Msg.hWnd`) to the handles of the application and main form window.

If the message received is the `WM_ToggleIcon` message then we need to change the Tray icon. The initial icon is a red letter Á; when compose mode is active this changes to a similar but green icon. The required icon is indicated by the `wParam` parameter of the message, so we check that, load the icon into the `NotifyIcon` structure, and pass it to the Tray with `Shell_NotifyIcon`.

If, instead, the message is `WM_TranslateKeys` then this means that a compose sequence has been completed. The two characters of the sequence are extracted from the `wParam` parameter using the `LoWord` and `HiWord` functions and passed to our `TranslateKeys` method, along with the handle of the currently focused window contained in `lParam`.

Translating Keystrokes

Translating the sequence is a simple process, as shown in Listing 7. We just scan through the string list comparing the two characters passed from the DLL to the third and fourth letters in each string (remember, these are preceded by the accented character and a tab). We check the compose sequence in either order, so that entering ‘"’ and ‘a’ gives the same character as ‘a’ and ‘"’. If a match is found the accented or extended character (the first character in the string) is copied to the local variable `AccChar`. This is then sent to the destined window (the handle of which was passed from the DLL) by posting a `WM_Char` message. If no match was found a beep

➤ *Listing 5*

```
procedure TDCompForm.InitDefaultList;
begin
  with CharList do begin
    Clear;
     Add('ß'+#9+'ss');
     Add('à'+#9+'a`');
     Add('á'+#9+'a''');
     Add('â'+#9+'a^');
     Add('ã'+#9+'a~');
     Add('ä'+#9+'a"');
     Add('å'+#9+'a*');
     Add('æ'+#9+'ae');
     Add('ç'+#9+'c,');
     Add('è'+#9+'e`');
     { ... }
end;
```

➤ *Listing 6*

```
procedure TDCompForm.AppOnMessage(var Msg: TMsg; var Handled: Boolean);
begin
  if (Msg.hwnd = Handle) or (Msg.hwnd = Application.Handle) then
    if Msg.Message = WM_ToggleIcon then begin
      if Msg.WParam = 1 then begin
        NotifyIcon.hIcon := LoadIcon(HInstance, 'LETTERGREEN');
      end else begin
        NotifyIcon.hIcon := LoadIcon(HInstance, 'LETTERRED');
      end;
      Shell_NotifyIcon (NIM_MODIFY, @NotifyIcon);
      Handled := true;
    end else if Msg.Message = WM_TranslateKeys then begin
      TranslateKeys(LoWord(Msg.wParam),HiWord(Msg.wParam),Msg.lParam);
      Handled := true;
    end;
end;
```

is issued and the two characters are sent to the window.

Delphi 1

So far we've written DCompose as a 32-bit program. Its reliance on the TaskBar tray restricts it to Windows 95 and NT 4. However that is just for the convenience of the user interface. The keyboard hook DLL can be compiled under 16-bit Delphi 1 with just a few changes; these are highlighted by `{$IFDEF Ver80}` compiler directives in Listing 2.

First, the required Windows units have different names, WinTypes and WinProcs. Also, we must declare the `wParam`, `lParam` and `lResult` types ourselves. In Delphi 2 and 3 they are declared in the Windows unit as 32-bit integers; however under 16-bit Windows the `wParam` parameter of messages is a 16-bit integer, so we declare that as a `word`.

There are slight differences in how some parameters are passed to the `ToASCII` and `SetWindowsHookEx` functions (the presence or absence of the `@` operator). Also, the functions and procedures in the DLL must be declared with the `export` directive rather than `stdcall`.

The most important change is in how the keys are passed to the parent program. Since the `wParam` message parameter is now a 16-bit word rather than 32-bit longint we cannot use the `MAKELONG` function to place the two characters in the lower and upper words of the parameter. Instead we make them the lower and upper bytes of the word parameter using the addition and shl operators:

```
PostMessage(HWND_BROADCAST,
  WM_TranslateKeys,
  word(FirstKey +
  (SecondKey shl 8)),GetFocus);
```

We must then modify the call to `TranslateKeys` in the DCompose application to extract the two bytes:

```
TranslateKeys(
  Lo(Msg.wParam),
  Hi(Msg.wParam),Msg.lParam);
```

There is one major functional difference between the 32-bit and 16-bit versions of the DLL to keep in mind. Earlier on I pointed out that under 32-bit Windows the DLL is mapped into the address space of each client application. As a result, each instance of the DLL has its own set of global variables.

Under 16-bit Windows 3.1 a single instance of a DLL is shared by all applications. This includes its global data. With the DCompose DLL this doesn't present too much of a problem. If you start a compose sequence while one application has the focus then finish it in another the character will show up in the window of the second application. Under 32-bit Windows DCompose will wait patiently until you switch back to the first program to finish the sequence before entering the character. Although this difference doesn't have much of an effect on this DLL with some others it could be a major problem.

Final Note

The translation scheme used in DCompose assumes that the standard Windows character set (based on the ANSI standard) is in use. This covers the characters commonly used in the Western European languages. However, it excludes many of the accented characters used in Eastern European languages. Of course, it also doesn't include the completely different alphabets like Cyrillic, Hebrew and Arabic.

If you are using the ANSI version of Windows then you often can't produce the eastern characters. However, many of the fonts designed for Windows 95/NT will have alternative character sets, such as Turkish or Central European, that replace ANSI characters with the ones of interest. For example, the character with the code 0200 is È in the default Windows character set, but a C hachek (a C with an inverted ^ mark) in the Baltic and Central European character sets. The same character can also have different codes: Ë is number 0203 in the Western character set, but 0168 in Cyrillic. If you wish to use DCompose with these alternative character sets then you will need to change the translation table accordingly. Of course, the text editing application will have to support the selection of different character sets as well; many don't.

The problems with different code pages for different languages disappears with the use of Unicode, a new standard that encodes thousands of the characters from different world languages (including numerous ideographs from Far Eastern languages). All characters can be produced on any system that supports Unicode. Unfortunately, Unicode is not fully supported in Windows 95, only Windows NT, and few mass market applications use it. Once Unicode is more widespread an enhanced version of a utility such as DCompose could be very useful for accessing the wider range of available characters.

When he isn't lurking behind your keyboard Warren Kovach writes and sells statistical software. He is also the author of *Delphi 3 User Interface Design*, published by Prentice Hall. You can contact him at wlk@kovcomp.co.uk

➤ *Listing 7*

```
procedure TDCompForm.TranslateKeys(First,Second : word;Wnd:THandle);
var i,AccChar : word;
begin
  AccChar := 0;
  for i := 0 to pred(CharList.Count) do begin
    if (CharList[i][3] = char(First)) and (CharList[i][4] = char(Second)) or
      (CharList[i][4] = char(First)) and (CharList[i][3] = char(Second)) then
      AccChar := ord(CharList[i][1]);
  end;
  if AccChar = 0 then begin
    PostMessage(Wnd,WM_Char,First,1);
    PostMessage(Wnd,WM_Char,Second,1);
    MessageBeep(-1);
  end else
    PostMessage(Wnd,WM_Char,AccChar,1);
end;
```