

East Meets East: Common Input Method Editor System on Far East Windows

Applications get more control; developers get greater flexibility

by Nadine Kano

Input method editors (IMEs), also called front-end processors, are applets that allow users to enter the thousands of different characters used in Far East written languages with a standard 101-key keyboard.

An IME consists of an engine that converts keystrokes into phonetic and ideographic characters, and a dictionary of commonly used ideographic words. As the user enters keystrokes, the IME engine attempts to guess which ideographic character or characters the keystrokes should be converted into. Conceptually, IMEs are the same as keyboard drivers—they simply handle more characters, and Far East systems require some extra code to handle them. When the IME program is active, it traps all keyboard events, including virtual keys.

Far East editions of Windows NT 3.5 and Windows 95 support popular input methods, such as phonetic, code point, or radical-based input methods. The Windows NT 3.5 IME model is compatible with the Windows 3.1 IME model, but both Windows 95 and Windows NT 3.51 implement a new IME model that makes application development simpler and more flexible.

The IME system

The IME module in the Windows operating systems fits into a larger mechanism. Unless you are writing an IME package or customizing your IME user interface (UI), you don't need to worry about the other components, but it is still a good idea to understand what they do.

In Windows NT, the input method profiler (IMP) keeps track of the IME applets installed in the user's system. It stores information about each IME, such as whether it is currently active. Control Panel calls the IMP API to add, delete, or activate an individual IME. More than one IME can be installed on the system, although with Windows NT 3.5, only one may be active at a time.

In Windows 95, multiple IMEs are handled through the multilingual API (instead of by an IMP). Windows stores information about each IME installed on the system in the system registry. Users switch IMEs the same way they would switch among Western keyboard layouts, by clicking the input language menu on the taskbar or entering a shortcut key combination. The change is reflected on the taskbar indicator.

The input method manager (IMM) manages communication between IMEs and applications. In Windows NT 3.5, the IMM API is called almost exclusively by the system. Applications can call two IMM functions: **WINNLSEnableIME**, which enables or disables an IME, and **WINNLSEnableStatus**, which returns the enabled or disabled status set by **WINNLSEnableIME**. In Windows 95, the IMM is an extension of USER.EXE, and fully IME-aware applications can call a number of IMM API functions to customize the IME UI.

IME support on Windows 95

With Windows NT 3.5, a single IME covers all subsystems, including Win32, Win16, console, virtual DOS machines, and OS/2. Implementing support for IMEs on Windows NT 3.5 involves adding code to handle a couple of window messages and adding calls to several APIs.

IME support on Far East editions of Windows 3.1 and Windows NT 3.5 differed slightly depending on the target language. The differences were inconvenient for developers; since IME-related code had to be customized depending on the target language, sharing a common code base among Far East editions of an application required additional work. The designers of Windows 95 therefore decided to create a common IME system for Far East editions of Windows 95 to give applications more control over mechanisms handled exclusively by the IME module on Windows 3.1.

For example, on Windows 95, applications now have the option of trapping and responding to messages before the IME processes them. Applications that contain code to support Windows 3.1- or Windows NT 3.5-based IMEs will still run correctly on Windows 95 (as long as the application executable is marked as being compatible with Windows NT 3.5 or an earlier version). But once you become familiar with the Windows 95-based model, you will probably want to rework your code to take advantage of the additional features offered, particularly since Windows NT 3.51 supports the same model.

Three levels of IME support

There are three discrete levels of IME support for applications running on Windows 95: no support, partial support, and fully customized support.

IME-unaware applications basically ignore all IME-specific window messages. Most applications that target single-byte languages will be IME-unaware. IME-unaware applications inherit the default UI of the active IME through a predefined global class, appropriately called IME. This global class has the same characteristics as any other Windows-based common control. For each thread, Windows 95 automatically creates a window based on the IME global class; all IME-unaware windows of the thread share this “default IME window.”

IME-aware applications

IME-aware applications can create their own IME windows instead of relying on the system default.

```
hIMEWnd = CreateWindowEx("IME", // IME class.
    NULL, // No window title.
    WS_DISABLED | WS_POPUP, // Disabled window.
    0, 0, 0, 0, // No need to set size.
    hWnd, // Parent window.
    (int)NULL,
    (HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE),
    NULL);
```

Applications with partial IME support can use this “application IME window” to control some IME behavior. By calling the **ImmIsUIMessage** function, applications can pass messages related to the IME’s UI to the application IME window, where the application can process them. The following code would appear in the application IME window’s window procedure.

```
If (ImmIsUIMessage(hIMEWnd, uMsg, wParam,
    lParam) == TRUE) {
    switch(uMsg) {
    case WM_IME_COMPOSITION:
    if (lParam & GCS_RESULTSTR) {
        hIMC = ImmGetContext(hWnd);
        ImmGetCompositionString(hIMC,
            GCS_RESULTSTR, lpBufResult, dwBufLen);
        ImmReleaseContext(hWnd, hIMC);
    }
    break;
    }
    return 0;
}
```

The same window procedure could call **SendMessage** to reposition the status, composition, or candidate windows, or to open or close the status window.

Other API functions that allow the application to change window positions or properties are **ImmSetCandidateWindow**, **ImmSetCompositionFont**, **ImmSetCompositionString**, **ImmSetCompositionWindow**, and **ImmSetStatusWindowPos**. Applications that contain

partial support for IMEs may use these functions to set the style and position of the IME UI windows, but the IME dynamic-link library (DLL) is still responsible for drawing them.

In contrast, applications that are fully IME-aware take over responsibility for painting the IME windows (the status, composition, and candidate windows) from the IME DLL. These applications can fully customize the appearance of these windows, including their positions on the screen and which fonts and font styles are used to display characters in them. This is especially convenient for programs such as word processors that rely heavily on text input and wish to interact with IMEs in a way that is as natural as possible for the user. The IME DLL still determines which characters are displayed in IME composition and candidate windows, and it handles algorithms for guessing characters and looking them up in the IME dictionary. An example of a customized IME UI is shown in Figure 1

(IMEFIG1.bmp goes here)

Fully IME-aware applications trap IME-related messages in the following manner:

1. They call **GetMessage** to retrieve intermediate IME messages.
2. They process these messages in the application **WindowProc**.
3. They call **TranslateMessage** (part of the IMM) to pass the messages on to the IME DLL. The IME needs to remain synchronized, the same way keyboard drivers need to remain synchronized with dead keys.

For example, when the user presses a key, the system generates a keyboard event. USER.EXE checks to see if the currently active HKL (input language handle) points to an IME. If so, it passes the keyboard event to the IMM, which passes the event to the IME. If the IME wishes to respond to the keyboard event, the IME will translate the keyboard event into the virtual key, VK_PROCESSKEY, which it passes back to the IMM. The IMM then sends this virtual key to the application.

If the application wishes to customize the IME UI, it should trap VK_PROCESSKEY. It calls **ImmGetVirtualKey** to translate the VK_PROCESSKEY into a more specific virtual key value, and responds accordingly.

IMEs on Windows 95 keep track of state information for each application running on the system. Each IME stores status information in an internal structure called an *input context*. Because Windows 95 creates and assigns a default input context to each new thread,

individual applications use separate input contexts. Within an application, each newly created window inherits the application's default input context, which is a shared resource.

It is possible to override the default input context by creating and maintaining your own input contexts. The next section describes how to do this.

Customizing IME support in Windows 95

While applications with partial support may reposition the IME's UI windows or change the fonts used to display characters, fully IME-aware applications create and maintain their own input contexts and draw the IME's status, composition, and candidate windows themselves.

Windows 95 provides a number of API calls for communicating closely with IMEs. Since IME DLLs still contain the code that converts keyboard strokes into characters and determines candidate lists, applications need to query IMEs for information on composition strings and candidate lists before they can draw any windows. Applications also need to ask IMEs for status information, such as whether the IME is open or closed and what the current input mode and sentence mode are.

To get or set IME status information, composition strings, or window positions, your application needs to send a handle to an input context to most API calls. Calling **ImmGetContext** will retrieve the default input context that the system has created and associated with all the windows of a particular thread. Once you are done with the input context, you need to release it by calling **ImmReleaseContext**. Be forewarned that the system's default input context is a shared resource—any changes to it will affect other windows in the same thread. To avoid this, you can create a custom input context—changes you make to a custom input context will only affect those windows you associate with it.

Before you associate a window with a custom input context, it's a good idea to save the window's default input context. You may need it later, for example, if you destroy your custom input context and need to restore the default.

To create an input context, call **ImmCreateContext**. To associate this context with a window, call **ImmAssociateContext**. Once you have associated an input context with a window, the system will automatically provide the input context when the window gets the focus. If you make changes to the input context, you should call **ImmNotifyIME** so that the IME can remain synchronized. To destroy the custom input context before you terminate your application, call **ImmDestroyContext**.

Fully IME-aware applications should handle most WM_IME messages. In general, the WM_IME messages tell the application that something has changed and that the application should query the system to get updated information. For more detailed information on these messages and the functions that your application should call in response to them, consult the FULLIME example that ships with the Far East Win32 SDK.

With the help of several API functions, you can be very creative in how you customize the UI for any or all of the IME windows.

If you plan to customize the candidate or composition windows, you should familiarize yourself with the **CANDIDATELIST**, **COMPOSITIONFORM**, and **CANDIDATEFORM** structures, plus the flags and style bits associated with these structures and related functions. The Win32 SDK documentation provides detailed descriptions.

For example, the **ImmGetCompositionString** function can give you a great deal of information about the string displayed in the composition window. It can tell you if the string contains the characters the user has entered, the characters resulting from the conversion, characters that have been selected but not yet converted, or characters that have been converted and are still selected. The function can also tell you which characters in the string constitute clauses and what position they hold in the composition string, where the cursor is positioned, and which characters the user originally entered.

If your application is very text intensive, it will probably benefit greatly from the extensive customization available with the new IME model. Applications that accept some text input will benefit from the functionality that Windows 95 and Windows NT 3.51 provide through partial support. Even applications that are completely IME-unaware will automatically inherit some IME functionality from the system. Developers and users alike will benefit from this new model—it's more consistent across all Far East editions of Windows and is thus easier to implement. In addition, it allows applications to integrate IME support in a creative and seamless fashion, making text input more intuitive for users.

This article is adapted from *Developing International Software for Microsoft Windows* by Nadine Kano. It is being published by Microsoft Press this spring.

Nadine Kano is a globalization guru in the Developer Relations Group. She has typed her fingers to the bone writing a guide to developing international software for Microsoft Press.

Conceptually, input method editors are the same as keyboard drivers—they simply handle more characters.

In Windows 95, multiple IMEs are handled through the multilingual API (instead of by an IMP).

IMEs on Windows 95 keep track of state information for each application running on the system.
