Legal Information

Programmer's Guide to Pen Services
For Microsoft® Windows® 95

Introduction

This book describes how to create applications that use the Microsoft® Windows® Pen Application Programming Interface (API). The book is divided into two parts. Part 1 presents an overview of pen-based computing and describes the various components of the Pen API. Sample code supplements the text and later chapters present a complete sample program and sample *recognizer* as examples. (Recognizers translate pen strokes into characters, symbols, or shapes.) Part 2 provides a reference for the functions, structures, messages, and constants that make up the Pen API. Following the reference, a number of appendixes provide information about the differences between versions 1.0 and 2.0 of the Pen API, the 32-bit pen services, and more.

The Microsoft Windows 95 operating system includes a subset of the Pen API for displaying pen data. This allows a pen-based application to collect pen data from a pen tablet, store the data, and later display the data on any personal computer running Windows 95, even without pen hardware. The full pen services come only with pen hardware from original equipment manufacturers (OEMs) of pen equipment. Thus, an application running with Windows 95 has guaranteed access to at least the display portion of the Pen API; if a pen tablet is attached, the application can also accept pen input.

The full pen services of the Pen API version 2.0 described in this book can run only with Windows 95 or later Windows versions.

This book assumes a familiarity with the C language and with Windows programming in general. To keep discussions concise, the text does not digress to define such general terms as *dynamic-link library* (DLL), *callback function*, or *message*. However, pen-based computing generates its own lexicon, so the text defines new terms specific to the Pen API as they are introduced. In addition, a brief glossary of terminology specific to pen-based computing appears at the end of this book.

Organization

This book is divided into the following chapters and appendixes.

Chapter or appendix	Describes
Chapter 1, Overview of the Pen Application Programming Interface	Architecture of the Pen API.
Chapter 2, Starting Out with System Defaults	How to add pen functionality to an application with a minimum of programming effort.
Chapter 3, The Writing Process	How an application gets input from a pen device.
Chapter 4, The Inking Process	How an application collects and changes pen input data.
Chapter 5, The Recognition Process	Converting raw pen input into usable characters such as letters and numerals.
Chapter 6, Design Considerations	Proper techniques, warnings, and tips for writing a pen-based application.
Chapter 7, A Sample Pen Application	The PENAPP.C sample application, to illustrate the information in Chapters 1 through 6.
Chapter 8, Writing a Recognizer	The requirements and design of a recognizer. Illustrates information using the sample recognizer SREC.C as a model.
Chapter 9, Summary of the Pen Application Programming Interface	Pen API services, listed by category.
Chapter 10, Pen Application Programming Interface Functions	Functions, listed alphabetically.
Chapter 11, Pen Application Programming Interface Structures	Structures, listed alphabetically.
Chapter 12, Pen Application Programming Interface Messages	Messages, listed alphabetically.
Chapter 13, Pen Application Programming Interface Constants	Constants, listed alphabetically.
Appendix A, Differences Between Versions 1.0 and 2.0 of the Pen Application Programming Interface	Changes and improvements to the Pen API.
Appendix B, Using the 32-Bit Pen Application Programming Interface	How to write 32-bit pen-based applications.
Appendix C, Modifying the SYSTEM.INI File	Settings used in Windows SYSTEM.INI.
Appendix D, Accessing the Pen Device Driver	How an application calls the pen driver.

Glossary Pen-based terms.

Document Conventions

The following document conventions are used throughout this book.

Convention	Description
Bold text	Bold letters indicate a specific term or punctuation mark intended to be used literally: language functions or keywords (such as **DrawPenDataEx** or **switch**), MS-DOS® commands, and command-line options. You must type these terms and punctuation marks exactly as shown. The use of uppercase or lowercase letters is usually, but not always, significant. For example, you can invoke the C compiler by typing either **CL**, **cl**, or **Cl** at the MS-DOS prompt.
( )	In syntax statements, parentheses enclose one or more parameters that you pass to a function.
Italic text	Italic text indicates a placeholder; you are expected to provide an actual value. For example, in the following syntax the placeholder *lpszRecogName* represents a pointer to the filename of a recognizer: **InstallRecognizer(** *lpszRecogName* **);** New terms pertaining to pen-based computing also appear in italics where they are first introduced or defined in the text. Such terms are also listed in the glossary.
Monospace text	Code examples are displayed in a nonproportional typeface.

if(!RegisterClass(LPWNDCLASS)&wc))
.
.
.
else
	A vertical ellipsis in a program example indicates that a portion of the program has been omitted.

. . .	A horizontal ellipsis following an item indicates that more items having the same form may appear.
[[ ]]	Double brackets enclose optional fields or parameters in command lines or syntax statements.
|	A vertical bar indicates that you can enter one of the entries shown on either side of the bar. In symbol

	graphs, a vertical bar indicates the possible character choices.
{ }	Braces indicate that you must specify one of the enclosed items.
SMALL CAPITAL LETTERS	Small capital letters indicate the names of keys and key sequences; for example, CTRL+ALT+DEL. If the key names are separated by commas instead of plus signs; for example ALT, F—then you must press the keys consecutively rather than together.

Books and Articles for Further Reading

The documentation listed in the following table provides additional information about the Pen API and about Windows in general.

Title	Contents
Microsoft Windows Pen API version 2.0 online Help	Online reference for Pen API functions, structures, messages, and constants.
Microsoft Windows Software Development Kit (SDK) documentation, or equivalent documentation	Information about the application programming interface of the Windows operating system.
Microsoft Windows Device Driver Kit (DDK) documentation, or equivalent documentation	Description of the application programming interface of the Windows device drivers. Required only for developing drivers.
Duncan, Ray. "Power Programming." *PC Magazine*. New York, New York: Ziff-Davis Publishing Company, January 14, 1992 through May 12, 1992	Series of articles on the basics of the Pen API version 1.0.
Petzold, Charles. *Programming Windows*. Third edition. Redmond, Washington: Microsoft Press, 1992	Good introduction to general programming for Windows.

System Requirements

You can develop pen applications for version 2.0 of the Pen API with the following software and hardware:

- A personal computer running Windows 95 or a later version of Windows
- A mouse, tablet, or other pointing device supported by the Pen API
- Microsoft Win32® Software Development Kit (SDK)
- Microsoft Windows 95 Device Driver Kit (DDK) − necessary only if you will be building pen, display, or keyboard drivers
- Microsoft C Optimizing Compiler, version 5.1 or later, or Microsoft QuickC® for Windows version 1.0 or later
- Microsoft Macro Assembler version 5.1 or later − necessary only if you will be building pen, display, or keyboard drivers

You may also use equivalent development software produced by other manufacturers, such as Borland International, Inc.

Acknowledgments

Special thanks to the contributors to this book including:

Writers	Beck Zaratian Don Gilbert	Mark Williams
Editors	David Steinmetz Peter Delaney	Barb Ellsworth David Thornbrugh
Program Managers	Jeff Aamodt Eric Berman	Steve Liffick
Pen Services Development Team	Eric Onasick Vinayak Bhalerao Haresh Ved	Shishir Pardikar Chris Leyerle Fumitaka Kawasaki
Recognition Development Team	Mike Van Kleeck Justin Ferrari Sung Rhee Donald Sidoroff Greg Hullender	Jim Adcock Shamik Basu Oswaldo Ribas Patrick Haluptzok
Testing Team	Becca Moss Brian Watson Xian-Ling Wu	Keith Stutler Randy Shedden David Flenniken

Overview of the Pen Application Programming Interface

This chapter presents an overview of pen-based computing, divided into two main sections. The first section broadly describes the various components that make up the Pen application programming interface (API). The second section describes how applications access the pen services to incorporate pen-based features.

The architecture of version 2.0 of the Pen API remains similar to version 1.0, but its style and design differ considerably. Even if you have worked with version 1.0, you should read this chapter to understand the shift in programming philosophy in version 2.0.

Architecture of the Pen API

The seemingly simple step of getting data from the pen to an application involves many intermediate tasks. Fortunately, the Pen API itself takes on the major share of this work. By providing applications with convenient access to pen features, the Pen API insulates the programmer from the most tedious aspects of pen data recognition. At the same time, its flexible design allows applications to control most of the low-level processes of pen input.

As you read this section, keep in mind that the complexities of the Pen API architecture in no way imply a corresponding difficulty in creating pen-based programs. You will find that writing intelligent pen-based software is no more difficult than writing other applications for Microsoft Windows.

Figure 1.1 illustrates the interaction between applications and the main components of the Pen API.

{ewc msdncd, EWGraphic, bsd23549 0 /a "SDK_1_1.BMP"}

The following four sections describe each component of Figure 1.1, beginning with the main Windows component. Each section contains a figure that incorporates Figure 1.1, exploding the component into a detailed view. The accompanying text describes the component and explains how it interacts with the other components.

Windows

As Figure 1.2 shows, the heart of the pen-based services for Microsoft Windows 95 consists of two libraries – PKPD.DLL and PENWIN.DLL. The PKPD.DLL file provides ink management for the pen services of Windows 95. This allows an application to display and manipulate ink data with any installation of Windows 95, even one without pen hardware. Chapter 9, "Summary of the Pen Application Programming Interface," identifies the pen services exported by PKPD.DLL.

{ewc msdncd, EWGraphic, bsd23549 1 /a "SDK_1_2.BMP"}

The PENWIN.DLL file is available only with original equipment manufacturer (OEM) pen hardware and provides additional pen services that collect, modify, and recognize ink data. Before using these input and recognition features, an application should first test for the presence of the PENWIN.DLL file and either gracefully exit or alter its behavior accordingly.

In Pen Windows version 1.0, applications were required to call **RegisterPenApp** in order to tell the system to convert all edit controls to handwriting edit (hedit) controls. With Pen API version 2.0, however, this is not necessary; all edit controls in applications are automatically converted. If the application is version-stamped as a Windows 95 - based application, the conversion is automatic; otherwise, applications version-stamped as Windows 3.1 - based applications require the call to **RegisterPenApp** that was required for Pen Windows version 1.0.

It is important to understand that for any application to successfully use the functions in PENWIN.DLL, the computer on which it is running must load the pen services when Windows boots and terminated the pen services when Windows shuts down (that is, PENWIN.DLLmust be referenced from the drivers line in the [Boot] section of the SYSTEM.INI file). This does not apply to functions in the PKPD.DLL library, which is automatically available on all Windows 95 systems. See Appendix C, "Modifying the SYSTEM.INI File," for information on the SYSTEM.INI file requirements.

Because of this requirement, PENWIN.DLL should never be statically linked by any application that may be run on a system on which pen services are not installed. Instead, its functions should always be called using function pointers. Typically, when the pen-aware application initializes, it calls **GetSystemMetrics** with **SM_PENWINDOWS** as a parameter which, if returned successful, provides a handle to the loaded library. Then, for each PENWIN.DLL function used by the application, the application calls the **GetProcAddress** function (with the library handle and the function name) and saves a function pointer to be used in future calls to that function. See the HFORM sample application for an example of this technique.

By not linking PENWIN.LIB, it is insured that an application running on a system on which PENWIN.DLL has not been installed, but which contains PENWIN.DLL on the path, will not load PENWIN.DLL at runtime. Pen components not loaded at system boot time are not guaranteed to perform properly. Note that this applies for both 16-bit and 32-bit libraries.

Applications that are destined to be run only on systems that have pen services installed can link directly to PENWIN.LIB. These applications should test for the existence of pen services at startup, however, and exit if it is not found. Note that most of the examples in this manual follow normal linking practice for the sake of readability and do not use the safer practice of using function pointers. It is the responsibility of the developer to choose the best means of accessing the functions in PENWIN.DLL for each application.

The Pen Message Interpreter provides basic pen services to *pen-unaware* applications. Such applications, which do not explicitly take advantage of Windows pen services, currently represent a majority of Windows-based software. The Message Interpreter allows use of a pen with pen-unaware programs by capturing handwritten input and other pen events and converting them into equivalent keyboard and mouse messages. The application has no knowledge of the pen or that pen input has occurred.

In capturing handwritten input, the Message Interpreter acts only when it detects a standard I-beam pointer or insertion point in the pen-unaware application. Since applications generally show the system I-beam pointer when prompting for input in writing areas, the Message Interpreter reliably serves most pen-unaware programs. However, a few pen-unaware Windows-based applications do not prompt with a standard I-beam pointer, defeating the Interpreter's detection method. Although the Interpreter still allows the pen to serve as a mouse with such applications, it cannot interpret handwritten input.

The Message Interpreter may also falter when serving applications developed for a version of Windows earlier than version 3.1. These applications were not designed with the pen in mind and therefore may not work optimally with the pen. For example, edit fields in applications written for Windows version 3.0 are often too small to write in with a pen. A final problem with older applications is that the Message Interpreter has no means of receiving contextual information from the application about what sort of input it expects. This can reduce recognition accuracy.

The Message Interpreter is of academic interest for the programmer because it pertains to only pen-unaware applications. The rest of this book focuses on how to write pen-aware applications and dynamic-link libraries (DLLs) that make use of the Pen API directly.

Drivers

Figure 1.3 shows the two types of drivers that function within the Pen API system. Most drivers incorporate two modules: an installable device driver that uses the Windows installable driver interface and a virtual device driver that handles interaction with the hardware.

{ewc msdncd, EWGraphic, bsd23549 2 /a "SDK_1_3.BMP"}

Pen Driver

The pen installable device driver, which Windows supplies as the file PENC.DRV, interacts with the virtual pen driver (VPENDC.VXD) and passes pen movement data to Windows. The fact that the pen driver's data may sometimes be needed for on-the-fly handwriting recognition places several constraints on a pen input device:

- The pen driver must be able to report the location of the pen at least 60 times per second. This rate ensures the true path of the pen is reported accurately enough to support the efforts of vector-based recognizers. It also makes the *ink*, a path of pixels that traces the pen's movement, appear smooth and natural at normal writing speeds. For more information about recognizers, see the "Recognizer" section later in this chapter.
- The pen driver must be able to report pen positions with a resolution of at least 200 points per inch. This degree of resolution ensures ink coordinates are sufficiently fine to let the recognizer make accurate judgments about the path of the pen over the digitizing surface.
- Regardless of the resolution of the device, the pen driver must report the pen position in tablet coordinates of 0.001 inch. This convention ensures that Windows, the recognizer modules, and the application all view the ink at the same scale.

Display Driver

The display driver is responsible for interacting with the display hardware and the graphics device interface (GDI) module of Microsoft Windows. A display driver should support inking to provide the user with visible feedback as the pen moves. Technically, the Pen API does not require inking support from the display driver. However, the system is far more practical and convenient when the user can see the ink trail left by the pen.

Two types of display drivers are supported: Display Control Interface (DCI) drivers (called *DCI Providers*) and non-DCI drivers, such as older VGA or 8514 drivers. For DCI Providers, no extra work is required to support the pen interface.

To support inking in a non-DCI driver, the display driver must be able to:

- Export the **GetLPDevice** function to provide Windows with a value identifying the pen hardware.
- Export the **InkReady** function, which Windows calls to notify the driver that the pen is in motion and Windows is ready to display ink. **InkReady** must be able to handle calls during system interrupts.
- Provide a pointer in the shape of a pen.

Windows – not the display driver – displays the ink. When Windows receives notice through its **InkReady** function, the driver calls back into Windows to draw the ink.

For more details on display drivers, refer to the device driver kit (DDK) for Windows 95.

Recognizer

A recognizer is a DLL with functions that determine what symbol a pattern of pen strokes represents. As illustrated in Figure 1.4, Windows allows the concurrent operation of more than one recognizer. For example, one recognizer may specialize in English letters, another in mathematical symbols, another in geometric shapes, and so forth.

{ewc msdncd, EWGraphic, bsd23549 3 /a "SDK_1_4.BMP"}

Each handwriting recognizer can access any number of *word lists*. Word lists offer a way for a recognizer to corroborate and refine its guesses. For example, if a recognizer cannot decide whether a handwritten word is "boy" or "looy," finding one word but not the other in a word list helps the recognizer make a more confident choice.

Although many recognizers may be available to an application, only one serves as the system default recognizer. This is the recognizer that Windows automatically installs and calls by default. To use other recognizers, an application must first specifically install them. (For information about how to install multiple recognizers, see Chapter 5, "The Recognition Process.") The Microsoft Handwriting Recognizer (GRECO.DLL) is provided as the default system recognizer on most OEM tablet installations of Microsoft pen services. The Microsoft Handwriting Recognizer recognizes all European letters, numerals, and punctuation, with emphasis on English, French, and German, An application can set up a different system recognizer by identifying the new file in the Windows registry. Appendix A explains how to set up a new default recognizer.

Accessing the Pen API from Applications

As Figure 1.5 shows, applications that accept user input are divided into two categories: pen-aware and pen-unaware applications. A pen-unaware application, as the name implies, is written to expect input only through the keyboard or mouse, unaware of the existence of Windows pen services. However, if a pen device is present, Windows 95 supports its use both as a mouse and for text entry with a pen-unaware application. For details about how Windows allows the use of a pen with an application not written to accept pen input, see "Pen-Unaware Applications" in Chapter 2, "Starting Out with System Defaults."

{ewc msdncd, EWGraphic, bsd23549 4 /a "SDK_1_5.BMP"}

The Pen API is designed for small handheld systems with limited memory and power, so its API consists of 16-bit functions. Therefore, Windows provides a *thunk layer* for 32-bit applications to call through to the API. The thunk layer automatically converts 32-bit function parameters and structure data to 16-bit equivalents. The application must ensure its data will fit into the smaller sizes before calling into the Pen API. See Appendix B for information about using the 32-bit API.

Starting Out with System Defaults

As much as possible, the Pen application programming interface (API) handles the many complexities of pen-based computing. A rich API leaves the developer free to concentrate on design without having to worry about details. This chapter describes how to create a pen-based application that relies on the system default services of the Pen API. For the sake of simplicity, the term "application" as used in this chapter refers both to Windows-based programs and dynamic-link libraries (DLLs).

Pen-Unaware Applications

Microsoft Windows 95 supports the use of a pen even with pen-unaware applications. For such applications, Windows provides a means for the pen to mimic both mouse and keyboard data. It does this in two ways.

The first method, the Pen Message Interpreter, is described in the "Windows" section in the previous chapter. The second method involves two utility "applets" called Writing Palette (WRITEPAL.EXE) and Screen Keyboard (SK.EXE), both supplied as installed applications. Writing Palette allows the user to enter handwritten text for those occasions when the Message Interpreter fails to detect an input prompt. For example, when running an MS-DOS text editor in a window, the user can input handwritten text through the Writing Palette utility. The Pen API translates the handwritten text into characters and displays the result in the writing window. The user can then correct the text if necessary and tap the OK button when the corrections are recognized. Windows feeds the characters to the pen-unaware text editor as a series of WM_KEYDOWN and WM_KEYUP messages as though they were typed at the keyboard.

The Screen Keyboard applet displays an image of a typical keyboard on which the user can "type" by tapping the keys of the *on-screen keyboard* with the pen. Each key is sent as soon as it is typed. This does not require recognition because no handwriting is involved.

Pen-Aware Applications

The Pen API allows the developer to approach pen-based computing in stages. For those who wish to do only a minimum amount of programming work and yet incorporate significant pen capabilities in an application, the Pen API provides the **DoDefaultPenInput** function. **DoDefaultPenInput** embodies a set of more complex API elements in one function. As its name implies, it allows applications to rely on the system to make all of the decisions concerning pen input. The developer can incrementally enhance a pen-based application as time and interest permit.

When called in response to a WM_LBUTTONDOWN message generated by the pen device, **DoDefaultPenInput** starts a cascade of messages. These messages reflect the many steps of the recognition process, each message serving as a notice that a next step in the process is about to occur. The application can take some action prior to each step or simply ignore the message and let the **DefWindowProc** function provide default services.

This approach follows standard Windows messaging procedures. If an application lets the message pass through to **DefWindowProc**, Windows translates the pen events into the appropriate keyboard messages. For example, handwritten characters generate appropriate WM_CHAR messages. In this way, a developer can gradually modify an existing application to become more and more sophisticated about pen input by adding code to handle more of the **DoDefaultPenInput** messages.

The following sections describe the programming convenience of using system defaults, which you might think of as "letting the system do the work." The text also mentions various options available to the developer who wishes to exercise more control over the recognition process. These options involve manipulating data objects such as **HRC** and **HPENDATA**, which are fully described in Chapters 4 and 5. The following sections serve as an introduction to the entire process of converting pen-based input to usable data. When you later decide to incorporate additional recognition management into your application, see Chapters 4 and 5.

Beginning an Input Session

A pen input session begins when the user touches pen to tablet and begins writing. The end of the session depends on parameters established by the application. Usually, the session ends when the user taps the pen outside the writing area or when a brief period of inactivity elapses. As when writing with a real pen, people tend to pause between words or sentences to gather their thoughts; an application can use these momentary pauses to get recognition results. A new session begins when the user begins writing again.

When the pen first touches the tablet at the start of an input session, Windows sends a WM_LBUTTONDOWN message to the application's main window procedure. In a pen-based environment, this message can indicate either a true mouse event or that the pen point has touched the tablet. The application must distinguish between these two possibilities before calling the **DoDefaultPenInput** function, as shown in the following fragment:

```
LONG lExtraInfo;
      .
      .
      .
switch (wMsg)
{
     case WM_LBUTTONDOWN:
           // If true pen-down event, call DoDefaultPenInput.
           lExtraInfo = GetMessageExtraInfo();
           if (IsPenEvent( wMsg, lExtraInfo )
                 return DoDefaultPenInput( hwnd, LOWORD(lExtraInfo) );

           else                                        // No, it's a mouse
           {                                                 //   button down
                 .
                 .
                 .
```

DoDefaultPenInput Messages

This section lists in chronologic order the message traffic that **DoDefaultPenInput** generates. It discusses why an application might want to handle each message and explains what action **DefWindowProc** takes. The sample application described in Chapter 7, "A Sample Pen Application," demonstrates how to handle most of these messages.

Step 1: PE_BEGININPUT Submessage

Immediately upon calling **DoDefaultPenInput**, an application receives a WM_PENEVENT message with a PE_BEGININPUT submessage. Sending WM_PENEVENT and PE_BEGININPUT via the **SendMessage** function is the equivalent of calling **DoDefaultPenInput**. The caller should not trap the PE_BEGININPUT submessage because **DefWindowProc** starts the chain of events based on this message. The application should complete all its initialization work before calling **DoDefaultPenInput**.

Step 2: PE_SETTARGETS Submessage

Windows sends the WM_PENEVENT message with a PE_SETTARGETS submessage to the window that received the PE_BEGININPUT submessage. PE_SETTARGETS is important when several windows on the screen vie for input at the same time, presenting Windows with more than one potential recipient for the pen data. This can occur when a dialog box contains multiple edit controls or a forms program prompts the user simultaneously with several writing areas. The user can write in different writing areas without having to pause between each and wait for recognition results. Windows treats the writing as part of a single input session, regardless of the targets.

DoDefaultPenInput must therefore select between *targets* when distributing pen data. A target is a rectangular area associated with the handle of a window that is a valid destination for pen data. When writing starts, all valid targets participate in the **DoDefaultPenInput** messaging. This allows the user to move freely between windows – for example, writing the name of a city in one control, interrupting to write the date in another control, then moving back to the first control to add the state and zip code. The system correctly routes pen input to the control on which ink was written or, barring that, to the control nearest the ink.

DoDefaultPenInput handles all routing automatically. Upon receiving a PE_SETTARGETS submessage, the application can process the message and create a **TARGINFO** structure that describes all valid targets for the pen data. If the application chooses not to process PE_SETTARGETS itself, **DefWindowProc** enumerates the children of the window and creates a **TARGINFO** structure automatically. If the application returns FALSE to the PE_SETTARGETS submessage, Windows assumes no targets exist and sends the pen data to the window that received PE_SETTARGETS.

For information on how to specify a target area larger than the window size, see the "PE_SETTARGETS Submessage"section.

Step 3: PE_GETPCMINFO Submessage

If the application calls **DefWindowProc** to process the PE_SETTARGETS submessage, every descendant of the window that received PE_SETTARGETS receives a PE_GETPCMINFO message. This message is so named because it gets information about the *pen collection mode* (PCM). The PCM describes the system state during an input session when the pen is writing and not operating as a mouse.

PE_GETPCMINFO gives each target the opportunity to:

- Proclaim or disclaim itself as a valid target.
- Specify termination conditions, such as timeout or range.
- Identify areas in which tapping terminates the input session.
- Do any combination of the above.

In processing PE_GETPCMINFO, the child window must fill in a **PCMINFO** structure that describes how pen interaction should proceed. If the candidate window wishes to receive input from the pen and become a true target, it can provide the coordinates of a bounding rectangle in the **rectBound** member of **PCMINFO**. The bounding rectangle constitutes the target area of the child window; inking that occurs within or nearest a bounding rectangle is sent to the window associated with the rectangle. If the child window does not process PE_GETPCMINFO, Windows does not consider the window a candidate for pen input but also does not prevent ink from overwriting the window.

DefWindowProc collects all bounding rectangles and exclusion rectangles provided by the descendant windows and creates a master **PCMINFO** structure that describes the situation.

For information about how to initialize and make changes to a **PCMINFO** structure, see the "Starting the Chain of Events" section in Chapter 3, "The Writing Process." See Chapter 11, "Pen Application Programming Interface Structures," for descriptions of the structure members.

Step 4: PE_GETINKINGINFO Message

Each target specified in the **TARGINFO** structure created in step 2 that has a valid bounding rectangle from step 3 receives a PE_GETINKINGINFO message. In response to this message, a child window can set ink color and ink width, establish the ink clip region, and specify whether or not Windows should automatically restore the screen and erase the ink after pen interaction has ceased.

Processing the message through **DefWindowProc** sets the system default ink attributes, uses the window boundary for the ink clip region, and forces automatic restoration of the screen after input. **DefWindowProc** merges the responses from each target into a master **INKINGINFO** structure.

Step 5: Master PCMINFO and INKINGINFO Structures

Having created a master **PCMINFO** structure and a master **INKINGINFO** structure, **DefWindowProc** sends one more PE_GETPCMINFO message and PE_GETINKINGINFO message to the parent window that contains the child targets. This provides the parent window a final opportunity to examine and change, if necessary, the system's assumptions about the impending inking event. For example, the parent window can specify a default ink color in the **INKINGINFO** structure or set an exclusion region around a screen object that had not, for some reason, handled the PE_GETPCMINFO message.

Step 6: PE_BEGINDATA Message

When pen activity destined for a particular target begins, the target first receives a PE_BEGINDATA message. This message provides the target a way to inform **DoDefaultPenInput** what to do with the data. If **DefWindowProc** handles this message, it assigns the pen data to a default **HRC** object and uses the system recognizer for recognition. (For more information about the system default recognizer, see "Recognizer" and "Creating the HRC.") Alternatively, the target can attach its own **HRC** for recognition, an **HPENDATA** to store the data, or a private object of some kind associated with the target.

To govern recognition, an application should handle PE_BEGINDATA, create and configure its own **HRC** object, and identify the object with the **dwData** member of the **TARGET** structure pointed to by the message's *lParam*. The application calls the **CreateCompatibleHRC** function to create the **HRC** object and set its context. This forces the system to use the new **HRC**. For more information about **HRC** and how to create one with **CreateCompatibleHRC**, see "The HRC Object."

Step 7: PE_MOREDATA Message

Multiple PE_MOREDATA messages can arrive at each target window to indicate more pen data is available. Generally, an application passes PE_MOREDATA on to **DefWindowProc** for default processing. **DefWindowProc** accrues new data by adding it to the **HRC** or **HPENDATA** object established in step 6.

Step 8: PE_ENDDATA Message

The PE_ENDDATA message informs a target window that input for the target has ceased. The message's *lParam* points to a **TARGET** structure, the **dwData** member of which identifies the **HRC** or **HPENDATA** created in step 6. If recognizing

input through an **HRC** object, the application should let **DefWindowProc** handle this message. However, **DefWindowProc** will destroy an **HPENDATA** object when processing PE_ENDDATA. To preserve the **HPENDATA**, the application has two choices:

- Trap the message, preventing it from reaching **DefWindowProc**. In this case, the **HPENDATA** object outlives the input session. Note that an **HPENDATA** object occupies memory in the system heap. When finished, the application must remove the object by calling **DestroyPenData** to avoid wasting resources.
- Alternatively, copy the **HPENDATA** object with **DuplicatePenData** before letting the PE_ENDDATA message fall through to **DefWindowProc**. However, this approach has no advantage over the preceding method, merely trading the original object for its clone. Again, the application is responsible for destroying the new **HPENDATA** object.

Step 9: PE_RESULT Message

The PE_RESULT message arrives only if the application has specified an **HRC** object in step 6, rather than an **HPENDATA** or other object. The message signals the target window that recognition results are ready. This message differs slightly from the others in that its *lParam* holds the **HRC** handle and not a pointer to a **TARGET** structure. If **DefWindowProc** handles PE_RESULT, it converts the recognizer's best guess to a string of characters and sends them to the target window as WM_CHAR messages. Gestures are also converted to appropriate messages, such as WM_COPY or WM_PASTE.

If the application handles the message, it must not destroy the **HRC** for the default system recognizer. Because **DoDefaultPenInput** created the default **HRC**, it expects to destroy it as well. The application must not destroy objects it did not create.

At this point, an application can process any of the results itself. For example, it might check for a recognized gesture such as the lasso or cut gesture. The procedure for examining gestures at this point involves three steps:

1. Retrieve any recognized gesture symbols from the **HRCRESULT** object by calling the **GetResultsHRC** function with the GRH_GESTURE argument.
2. If this call indicates the recognizer has found a gesture, the application then calls the **GetSymbolsHRCRESULT** function to see if the gesture is a lasso or X mark.
3. If the gesture is a lasso or X mark, the application should examine the data further to determine the size of the gesture, as outlined in the following example.

If the first or second test fails, indicating the recognizer has found no lasso or X mark, the application should pass the PE_RESULT message to **DefWindowProc** for text processing. Note that the lasso and cut gestures cannot exist with other gestures; therefore, the following code allocates only one **HRCRESULT** object because it examines at most a single gesture:

```
HRCRESULT  hresult;                              // Look at only the first
gesture
HPENDATA         hpendata;                       // Points that comprise the
gesture
HRGN       hrgn;                           // Screen region of the gesture
SYV              syv;                            // Symbol value of the
gesture
UINT       uRgnType;                       // Region type: X or LASSO
int              cGest;                                // Count of gestures in
results
     .
     .
     .
switch ( wParam )                                // Handle WM_PENEVENT
messages
{
     case PE_RESULT:
          // Check for gesture
          cGest = GetResultsHRC( (HRC) lParam,                  // HRC
handle
                                                 GRH_GESTURE,                //
Gestures only
                                                 (LPHRCRESULT)&hresult,      //
Buffer
```

```
                                          1 );
     // Get one result

           // If one gesture available, get its symbol
           if (cGest == 1)
           {
                 GetSymbolsHRCRESULT( hresult,       // HRCRESULT handle
                                                 0,                      //
Index to 1st syv
                                                 (LPSYV) &syv,    // Symbol
buffer
                                                 1 );             // Get 1
symbol

                 //
                 // If the gesture is lasso or x, collect the
                 // points that make up the gesture
                 //
                 if (syv == SYV_LASSO  ||  syv == SYV_CUT)
                 {
                       hpendata = GetPenDataHRC( (HRC) lParam );
                       if (hpendata)
                       {
                             // Step 1:  Get region of the gesture
                             uRgnType = (syv==SYV_LASSO) ? CPDR_LASSO :
CPDR_BOX;
                             hrgn = CreatePenDataRegion( hpendata,
uRngType );


                             // Step 2: Determine what text lies within the
                             // region.  If the gesture covers more than one
                             // letter of a word but not the entire word,
assume
                             // it's meant for entire word.
                                   .
                                   .
                                   .
                             // Step 3:  Either select . . .
                             if (syv == SYV_LASSO)
                             {
                                   .
     // Select the text
                                   .
                                   .
                             }
                             // . . . or delete the text
                             else
                             {
                                   .
     // Delete the text
                                   .
                                   .
                             }
                             DeleteObject( hrgn );
```

```
                        DestroyHPENDATA( hpendata );
                    }
                }
            }
            break;

      default:
            DefWindowProc( hwnd, message, wParam, lParam );

}
```

As the previous code shows, applying a gesture to text requires three steps:

1. Call the **CreatePenDataRegion** function to find the region covered by the gesture.
2. Determine the text that lies within the gesture's region using the Windows **GetTextExtentPoint32** function or some other method.
3. Select or cut the text, according to the gesture.

Step 10: PE_ENDINPUT Message

Windows sends the PE_ENDINPUT message to the window that received the PE_BEGININPUT submessage in step 1. An application can perform any necessary cleanup chores at this time, but should pass the PE_ENDINPUT message to **DefWindowProc**.

The Writing Process

This chapter begins a series of three chapters that describe the three main stages of converting pen input into valid computer data. This chapter looks at the writing process. The next two chapters discuss the processes of inking and recognition.

The writing process includes the various ways a user can write input to a pen-based application. These involve not only writing words and scribbling figures with a pen, but also gesturing with predefined pen movements and tapping an on-screen keyboard.

This chapter is divided into three main sections, each discussing a different method by which an application can accept writing from the pen. The first section describes the pen edit controls, which are pen-based versions of a standard Windows edit control. The second section discusses *ink input* application programming interface (API) services, which allow an application to govern pen interaction at a lower level than edit controls. The final section briefly describes the on-screen keyboard.

Pen Edit Controls

The Pen API provides three different edit controls for pen input – the handwriting edit (hedit), boxed edit (bedit), and ink edit (iedit) controls. (The first letter of the control name is pronounced separately, as in "h-edit.")

The first two controls are designed for text input. Characters written in an hedit or bedit control are passed to one or more recognizers and interpreted as equivalent digital text. Usually, the interpreted text replaces the handwritten glyphs within the control window after the recognition finishes. The iedit control serves as a drawing area. With the exception of gestures, the iedit control does not attempt to recognize written input but merely preserves the pen data in its raw form.

An application creates a pen edit control through the Windows **CreateWindow** function, specifying a class of HEDIT, BEDIT, or IEDIT. The next three sections describe the pen edit controls in detail and provide examples of how to create them.

The hedit Control

Except under special circumstances, the hedit control displays two sets of text. First, the handwritten characters appear as written by the user, formed by the ink trail of the pen. When the writing is recognized, the handwritten ink disappears from the screen, replaced within the control window by the interpreted characters as determined by the recognizer. The interpreted text appears in a Windows font as though typed at the keyboard.

The following instruction creates a multiline hedit control with left justification:

```
hwndHedit = CreateWindow( "HEDIT", NULL,
                          ES_MULTILINE | ES_LEFT |
                          WS_CHILD | WS_VISIBLE,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          hwndParent, CHILD_ID, hinstCurrent, NULL );
```

The styles of ES_MULTILINE and ES_LEFT do not constrain the freeform approach of handwriting in an hedit control. The user can write anywhere on the pen tablet allowed by the control – usually within or near the control window. The styles determine the format of the resulting interpreted text displayed in the control.

The hedit control is a pen-aware version of the default Windows 3.1 edit control. The hedit control not only supports handwritten characters and gestures, but also responds normally to keyboard and mouse events in the same way as an edit control. An application can use an hedit control anywhere a regular edit control will work, including dialog boxes. In fact, specifying EDIT class for a control in Windows 95 automatically creates an hedit control. The hedit control is visually identical to a standard control except that it displays a pen pointer instead of an I-beam pointer. Single-line hedit controls also display a lens button when a pen is present. When specifying an HEDIT control for a dialog box in a resource, use the **DIALOGEX** resource. Refer to the DIALOGEX resource description in the Win32 SDK tools documentation for a more information on using the HEDIT control class.

With a window style of ES_READONLY, an hedit window can accept no pen input. The pointer within the control does not change to a pen.

An hedit control processes tabs and carriage returns differently depending on whether they are entered as gestures or typed from the keyboard. If the user draws a tab or carriage return gesture in a multiline hedit control, the control inserts a tab or carriage return character into the text. For tabs and carriage returns entered from the keyboard, an hedit control in a dialog box mimics the standard dialog box behavior – that is, pressing the TAB or RETURN key passes control respectively to the **TABSTOP** or **DEFPUSHBUTTON** statement given in the dialog box template.

hedit Control Messages

The Pen API defines the WM_PENCTL message and its alias, WM_HEDITCTL. An application can send the WM_PENCTL message to an hedit control like this:

```
lRet = SendMessage( hwndHedit, WM_PENCTL, wParam, lParam );
```

The *wParam* parameter of WM_PENCTL contains an identifier for an HE_ submessage, as listed in Chapter 12, "Pen API Messages." The *lParam* specifies a value dependent on the HE_ submessage. For more information about the *wParam* and corresponding *lParam* values, see the entry for WM_PENCTL messages in Chapter 12, "Pen Application Programming Interface Messages."

Sizing the Writing Area with Control Messages

An hedit control must make allowances for handwriting input by providing a sufficiently large area in which to write. Typically, this area incorporates the control window itself plus an ample margin around the border of the window. Besides increasing user comfort, this extra space helps ensure parts of written characters are not inadvertently clipped, making them difficult to recognize. For example, the cut gesture X often extends above the text selected for deletion. Losing part of the gesture at the edge of the control window can make it less recognizable.

Note that adjusting a control's writing area does not change the appearance or size of the control window on the screen. It only specifies an invisible area overlaying the window; any ink within the writing area belongs to the control. It is possible, though not recommended, to enlarge the writing areas of two nearby controls so that they overlap. In this case, Windows assumes ink within the overlapping area belongs to only one of the control windows, according to normal Windows z-ordering.

The Pen API provides two methods for an application to adjust the size of the control writing area. These methods involve either receiving a PE_SETTARGETS submessage or sending an HE_SETINFLATE submessage. The following sections describe both methods.

PE_SETTARGETS Submessage

As described in the "DoDefaultPenInput Messages" section in Chapter 2, Windows sends a PE_SETTARGETS submessage to the application's window procedure before ink collection begins. This submessage gives the application the opportunity to set the target writing areas by specifying a new **TARGINFO** structure identified by *lParam*. The structure member **rgTarget** contains an array of **TARGET** structures, one for each target area. The rectangle in the **rectBound** member of **TARGET** specifies each target's writing area. The following code fragment shows how to set a writing area 4 pixels larger than the boundaries of the child window:

```
#define      NTARG     3               // Number of target windows
#define      MARGIN    4               // Inflation margin in pixel units

LPTARGINFO   lpti;                   // Allocate new TARGINFO structure
HWND         hwndCtl[NTARG];         // Handles to child windows
RECT         rect;                   // Bounding rectangle of child
RECTL        rectl;                  // Long version of bounding rect
HGLOBAL      h;
     .
     .
     .
h = GlobalAlloc( sizeof( TARGINFO ) + (NTARG - 1)*sizeof( TARGET ) );
lpti = GlobalLock( h );
lpti->cbSize = sizeof( TARGETINFO );
lpti->wFlags    = 0;
lpti->htrgOwner = HtrgFromHwnd( hWnd );
```

```
lpti->cTargets  = NTARG;

for (i=0; i < NTARG; i++)
{
     GetWindowRect( hwndCtl[i], (LPRECT) &rect );
     rectl.left   = (LONG) (rect.left   - MARGIN);    // Inflate
     rectl.top    = (LONG) (rect.top    - MARGIN);    //   rectangle
     rectl.right  = (LONG) (rect.right  + MARGIN);    //   by MARGIN
     rectl.bottom = (LONG) (rect.bottom + MARGIN);    //   pixel units
     lpti->rgTarget[i].idTarget         = i;
     lpti->rgTarget[i].htrgTarget       = HtrgFromHwnd( hwndCtl[i] );
     lpti->rgTarget[i].rectBound.left   = rectl.left;
     lpti->rgTarget[i].rectBound.right  = rectl.right;
     lpti->rgTarget[i].rectBound.top    = rectl.top;
     lpti->rgTarget[i].rectBound.bottom = rectl.bottom;
}
```

If the Windows **DefWindowProc** function handles the PE_SETTARGETS submessage, it creates a **TARGINFO** structure identifying all child windows as targets. **DefWindowProc** does not inflate writing areas; that is, it sets the writing area for each child window within the window borders.

HE_SETINFLATE Submessage

An application can also enlarge a control's writing area by sending the submessage HE_SETINFLATE to the control window specifying a **RECTOFS** structure:

```
typedef struct {
                   int  dLeft;            // Left margin
                   int  dTop;             // Top margin
                   int  dRight;           // Right margin
                   int  dBottom;          // Bottom margin
                } RECTOFS FAR * LPRECTOFS;
```

The **RECTOFS** structure does not contain the coordinates of a writing rectangle per se; instead, it contains the dimensions of the additional writing margin around the control window. The margins specify how many pixel units to add to each member of the windows rectangle. Margins conform to the x-y screen coordinate system. Thus, to inflate a writing area, specify negative values for **dLeft** and **dTop** as shown here:

```
#define     MARGIN    4              // Inflation margin in pixel units
RECTOFS     rectofs = { -MARGIN,     // Structure of window margins
                        -MARGIN,
                         MARGIN,
                         MARGIN};
     .
     .
     .
wParam = HE_SETINFLATE;
lParam = (LONG)((LPRECTOFS) &rectofs);
lRet   = SendMessage( hwndHedit, WM_PENCTL, wParam, lParam );
```

An application can retrieve a window's current inflation margins with the submessage HE_GETINFLATE like this:

```
wParam = HE_GETINFLATE;
lParam = (LONG)((LPRECTOFS) &rectofs);
```

```
lRet   = SendMessage( hwndHedit, WM_PENCTL, wParam, lParam );
```

This call fills the **RECTOFS** structure pointed to by *lParam* with the control window's current margins.

Notification Messages

An hedit window's parent receives the same EN_ notification messages as the parent of a standard edit window. The parent receives a WM_COMMAND message in which the low-order word of the *wParam* parameter contains the control ID number and the *lParam* parameter contains the edit window handle. In 16-bit applications, the high-order word of *lParam* also contains the notification value. In 32-bit applications, the high-order word of *wParam* contains the notification. The hedit control also provides HN_ notifications, described in Chapter 12, "Pen Application Programming Interface Messages."

The hedit control also sends a WM_CTLINIT message to its parent windows when created. The *wParam* parameter holds the constant CTLINIT_HEDIT and *lParam* points to a **CTLINITBEDIT** structure. The structure contains the current system assumptions concerning the appearance and behavior of the hedit control. The parent window has the option of changing any of these assumptions.

Printing an Edit Control

An application can display a pen edit control in any device context by sending the control a WM_PRINT message. The message's *wParam* contains the **HDC** handle for the device context. This technique, which applies to all pen edit controls, provides a means for printing the contents of a control window.

The bedit Control

The bedit (boxed edit) control is a variation of the hedit control. All characteristics of an hedit control described in the preceding section also apply to the bedit control, with two exceptions:

- A bedit window displays writing guides in which the user must write. Interpreted text returned from the recognizer replaces the handwritten characters within the guides.
- A bedit control does not support edit control styles ES_READONLY, ES_CENTER, ES_LEFT, and ES_RIGHT. (Text in a bedit is left aligned.)

An application can specify guides either as a *comb* or as a set of boxes, as shown in Figure 3.1. The comb consists of a horizontal line with spaced tick marks. The user writes individual characters between the marks.

{ewc msdncd, EWGraphic, bsd23550 0 /a "SDK_02.BMP"}

These visual guides can greatly improve recognition because they remove from the recognizer the significant burden of correctly segmenting the text into separate characters. For example, consider Figure 3.2, which shows a word written in an hedit control.

{ewc msdncd, EWGraphic, bsd23550 1 /a "SDK_03.BMP"}

In this case, the recognizer would probably have difficulty choosing between the words "clean" and "dean" because of the narrow spacing between the first two strokes. The bedit control removes such ambiguities. By writing within the guides of a bedit control, the user implicitly informs the recognizer what strokes compose a single character.

For a description of how the bedit control has been improved in Windows 95, see Appendix A, "Differences Between Versions 1.0 and 2.0 of the Pen Application Programming Interface."

Windows treats characters in a bedit control as a continuous stream of text. If the control contains more than one row, text wraps at each row end without regard to word boundaries. The EM_SETWORDBREAK message has no effect on a boxed edit control.

An application creates a bedit control with the Windows **CreateWindow** function, specifying a window class of BEDIT. The following code shows how to create a multiline bedit control:

```
hwndBedit = CreateWindow( "BEDIT", NULL,
                          ES_MULTILINE | WS_CHILD | WS_VISIBLE,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          hwndParent, CHILD_ID, hinstCurrent, NULL );
```

The **BOXLAYOUT** structure governs the height and style of the box grid within the bedit control. Its **style** member accepts one of the following BXS_ values:

Style	Description
BXS_NONE	Resets current box style to the default comb style.
BXS_RECT	Specifies a grid of closed rectangular boxes.
BXS_BOXCROSS	Specifies small crosses at the center of each box. This is used mainly to aid recognition of certain Far Eastern languages.

To set box height and style, fill a **BOXLAYOUT** structure with the desired values and pass it as a submessage of WM_PENCTL, as shown here:

```
BOXLAYOUT boxlayout;
        .
        .
        .
boxlayout.cyCusp    = 6;             // Box sides are 6 pixels high
boxlayout.cyEndCusp = 6;             // Ends should be the same
boxlayout.style     = BXS_RECT;      // Grid of closed boxes
iRet = SendMessage( hwndHedit, WM_PENCTL, HE_SETBOXLAYOUT,
                    (LONG)((LPBOXLAYOUT) &boxlayout) );
```

Refer to Chapter 11, "Pen Application Programming Interface Structures," for a more detailed description of the **BOXLAYOUT** structure. The next section explains how to set the system assumptions about the appearance of a bedit control.

bedit Control Messages

The bedit control sends a WM_CTLINIT message to its parent windows when created. The *wParam* parameter holds the constant CTLINIT_BEDIT and *lParam* points to a **CTLINITBEDIT** structure. The structure contains the current system assumptions concerning the appearance and behavior of the bedit control. The parent window has the option of changing any of these assumptions.

An application can also initialize a bedit control for a dialog box by using a **DIALOGEX** resource in the dialog resource file (.RC) and specifying a **CONTROL** statement with a class of BEDIT or specifying a BEDIT edit control class. In this case, the control still sends the WM_CTLINIT message. However, the **CTLINITBEDIT** structure reflects the specifications of the BEDIT class statement instead of the system defaults. As before, the parent window can modify the structure if desired. Refer to Chapters 11 and 12 for descriptions of the **CTLINITBEDIT** structure and WM_CTLINIT message, respectively. Refer to the **DIALOGEX** resource description in the Win32 SDK tools for more information on using the BEDIT control class.

The EM_LIMITTEXT message deserves special mention because it has changed slightly from version 1.0 of the Pen API. The message now sets the maximum number of bytes of text, rather than the number of boxes, that the control can hold. Note that although a newline character occupies only one box, the newline itself – carriage return and linefeed – takes 2 bytes. Certain Far Eastern languages also require 2 bytes per character.

Thus, the EM_LIMITTEXT message has the same effect on bedit controls as it does on hedit and edit controls. For example, the instruction

```
SendMessage( hwndBedit, EM_LIMITTEXT, 50, 0L );
```

sets to 50 the number of bytes the bedit control can accept. This has the following effects on the control:

- If the user attempts to write the 51st byte, the control beeps and ignores the input.
- If the user inserts text into existing text, the control beeps and ignores further input after the total number of bytes equals 50.

Using bedit Controls in Dialog Boxes

Windows determines the number of box cells that can fit within a control window based on the window dimensions and the cell widths given in a **GUIDE** structure. Although the Pen API does not provide a way to explicitly set the number of boxes displayed in a bedit control, an application can imply the number by adjusting the size of the control window or the size of the cells. Under certain circumstances, however, Windows may change the dimensions of a bedit control in a dialog box, thus potentially decreasing or increasing the number of box cells within the bedit.

Usually, this makes no difference to the developer or the user. But if your application must always show a specific number of boxes within a bedit, this section explains how to forestall or handle any changes.

By default, Windows sizes a dialog box and its controls based on the system font. If the dialog template requests a different font with a **FONT** statement and the font is not available when Windows creates the dialog box, Windows selects an available font that best matches the requested font. It then scales the dialog box and the controls within it according to the size of the selected font, but does not also scale the bedit guides. Thus, although a bedit window may change in size because of new scaling, the size of the boxes within it remain the same. For this reason, the window may end up with fewer or more cells than the programmer expects.

To ensure a bedit window always displays a specific number of cells, use one of the following techniques:

- Remove the **FONT** statement from the dialog template and let Windows use the system font in the dialog box controls.
- Specify a font likely to be always available. However, this technique cannot guarantee correct results for an application that must run under many different configurations of Windows.
- Readjust the size of the bedit window after Windows has changed it. When processing WM_INITDIALOG, an application can call the **GetWindowRect** function to see whether Windows has resized the bedit control window. If so, the application can restore the window's original size with either the **MoveWindow** or **SetWindowPos** function.

  Note that this technique assumes a generous blank area surrounds the control, so that if your application enlarges the control window while Windows shrinks the rest of the dialog box, the various components of the dialog box do not overlap.
- Recalculate the **GUIDE** values if Windows has changed the window size.

By default, all the controls within a dialog box use the font selected for the dialog box. An application can set a different font in bedit controls within the dialog box by sending a WM_SETFONT message when processing WM_INITDIALOG.

The iedit Control

The ink edit (iedit) control provides easy formatting and manipulation of ink input. It is not designed for text input, and in this regard differs from the other two pen edit controls, hedit and bedit. Think of iedit instead as a convenient drawing area suitable for sketches, diagrams, signatures, doodling – any sort of pen input that does not need to be recognized as text. However, an application can collect handwritten text as input from an iedit control and later transfer it to an hedit or bedit control for editing, if desired, or send it to a recognizer for recognition.

An iedit control ignores most keyboard input because the user cannot type text into an iedit window. However, an iedit control supports the following keystrokes and key combinations as convenient shortcuts:

Keystroke or key combination	Effect
DEL	Delete selected strokes.
CTRL+X	Cut selected strokes to clipboard.
CTRL+C	Copy selected strokes to clipboard.
CTRL+V or CTRL+P	Paste stroke from clipboard
CTRL+A	Select all strokes.
CTRL+Z	Undo last command.

An iedit window can scroll like any other edit control. Specifying a window style incorporating WS_VSCROLL and WS_HSCROLL creates a scrollable drawing area of 32,767 by 32,767 coordinate units. Scroll bars appear on the iedit window only when ink resides outside the current visible area. This behavior mimics the Control Panel window, which displays scroll bars only when an icon lies hidden beyond the boundaries of the window.

The following sample procedure demonstrates how to use iedit to create a drawing area within a single window. After creating the main parent window, the procedure InitInstance copies the window's coordinates into a **RECT** structure. It then uses the results when sizing the child iedit window so that the child window entirely overlays its parent.

```
HWND    vhwndMain;                  // Main window
HWND    vhwndIedit;                 // iedit control window
     .
     .
     .
BOOL InitInstance( HINSTANCE hInstance, int nCmdShow )
{

    RECT    rect;                   // Main window rectangle

    //
    // Create main window
    //
    vhwndMain = CreateWindow(
        "StylusClass",              // Window class name
        "Stylus Sample Program",    // Text for title bar
        WS_OVERLAPPEDWINDOW,        // Window style
        CW_USEDEFAULT,              // Default horizontal position
        CW_USEDEFAULT,              // Default vertical position
        CW_USEDEFAULT,              // Default width
        CW_USEDEFAULT,              // Default height
```

```
        NULL,                       // No parent
        NULL,                       // Class default menu
        hInstance,                  // Window owner
        NULL );                     // Unused pointer

    if (!vhwndMain)                 // If can't create window,
        return FALSE;               //   exit

    //
    // Create iedit control window within main window
    //
    GetClientRect( vhwndMain, (LPRECT) &rect );

    vhwndIedit = CreateWindow(
        "IEDIT",                    // Window class name
        NULL,                       // No title bar
        WS_CHILD | WS_VISIBLE |     // Window style
        WS_HSCROLL | WS_VSCROLL,
        0,                          // Overlay control window
        0,                          //   onto parent window
        rect.right - rect.left,     // Use parent width
        rect.bottom - rect.top,     //   and parent height
        vhwndMain,                  // Parent window handle
        (HMENU) CHILD_ID,           // Child ID
        hInstance,                  // Window owner
        NULL );                     // Unused pointer

    if (!vhwndIedit)                // If problem,
        return FALSE;               //   return error code

    SetFocus( vhwndIedit );           // Give control immediate focus
    ShowWindow( vhwndMain, nCmdShow ); // Display main window
    UpdateWindow( vhwndMain );        // Force WM_PAINT message

    return TRUE;                      // Return success
}
```

iedit Control Messages

When created, an iedit control behaves similarly to a bedit control, as described in the "bedit Control Messages" section. The iedit sends a WM_CTLINIT message to its parent window. The message's *wParam* parameter contains the constant CTLINIT_IEDIT and *lParam* points to a **CTLINITIEDIT** structure. The structure contains the current system assumptions concerning the appearance and behavior of the iedit control.

An application can initialize an iedit control in a dialog box by specifying the desired attributes in an IEDIT statement in the .RC file. See "bedit Control Messages" for details. For information about the **CTLINITIEDIT** structure, see Chapter 11.

Ink Input

The pen edit controls discussed previously provide a simple and efficient method for an application to accept handwritten input. Pen edit controls continue the philosophy and design of a standard Windows edit control; that is, they place the burden of getting user input on the system rather than the application.

However, ink input API services also offer an application low-level control over the writing process. Ink input allows an application to gather raw data from the pen, then process it in any way it wishes. For example, the application can manage its own inking or even postpone inking to a later time. It can massage or filter the pen data in some way – say, by rotating an image based on pen movement. It can pass the data to a handwriting recognizer or simply throw the data away. Ink input offers an application greater freedom with ink data than simply parsing it for characters.

As you might expect, the increased control afforded by ink input requires increased programming effort. The flexibility of ink input does not allow a simple recipe of tasks, but in broad terms the three main steps are as follows:

1. Start the chain of events.
2. Collect and display data.
3. Process the data.

An application can rely on the **DoDefaultPenInput** function to collect and process ink input. For a description of this function, see Chapter 2, "Starting Out with System Defaults." The following sections focus on the lowest-level API services. Through these low-level services, an application has complete control over ink input. These are the same services **DoDefaultPenInput** calls internally. If you have read Chapter 2, the message traffic described here will seem familiar.

The PENAPP sample application described in Chapter 7, "A Sample Pen Application," demonstrates how to use the low-level API services for ink input. Most of the code fragments in the following sections appear in the PENAPP.C source listing located in the SAMPLES\C\PENAPP directory.

Starting the Chain of Events

When an application receives a WM_LBUTTONDOWN message, that message signals the beginning of an input session. (For a definition of input session, see "Beginning an Input Session" in Chapter 2.) The following code fragment shows how to handle the message when not calling **DoDefaultPenInput**:

```
INKINGINFO   inkinfo;
PCMINFO      pcminfo;
HPCM         hpcm;
LONG         lExInfo;
int          iRet = TRUE;
     .
     .
     .
switch (wMsg)
{
     case WM_LBUTTONDOWN:                                        // Pen
down (maybe)
          lExInfo = GetMessageExtraInfo();

          if (IsPenEvent( wMsg, LOWORD(lExInfo) )   // If true pen down,
          {                                                          //
take the actions
               .                                                     //
described in the
               .                                                     //
text below
               .
          }
          else                                                  // No,
it's a mouse
          {                                                          //
button down
```

The application initializes a **PCMINFO** structure and **INKINGINFO** structure in preparation for data collection and display. **DoDefaultPenInput** handles this task automatically, then provides the application an opportunity to selectively change the default values. (For a description of the submessages PE_GETPCMINFO and PE_GETINKINGINFO, see "DoDefaultPenInput Messages" in Chapter 2.) If it does not call **DoDefaultPenInput**, an application must initialize the structures itself, as shown here:

```
#define         TIME_OUT      1000              // Time-out in msec
PCMINFO         pcm;
INKINGINFO      ink;
     .
     .
     .
pcm.cbSize = sizeof( PCMINFO );                 //  Required
pcm.dwPcm = PCM_TIMEOUT | PCM_RECTBOUND;        //  When to end session
pcm.dwTimeOut = TIME_OUT;                       //  Time-out value in ms
GetWindowRect( hwnd, (LPRECT) &pcm.rectBound ); //  Use window borders
ink.cbSize    = sizeof( INKINGINFO );           //  initialize
     .                                           //    the INKINGINFO
     .                                           //    structure "ink"
     .
```

The above lines specify that the input session ends when the pen travels (or taps) outside the hwnd window or pauses for the number of milliseconds set by the constant TIME_OUT. For a detailed description of these structures, see the entries for **PCMINFO** and **INKINGINFO** in Chapter 11, "Pen Application Programming Interface Structures."

Collecting and Displaying Data

After initializing the necessary structures, the application calls **StartPenInput** to begin the process of collecting ink data:

```
hpcm = StartPenInput( hwnd, LOWORD (lExtraInfo),
                      (LPPCMINFO) &pcm, NULL );
```

The returned value hpcm is a handle to the pen collection mode – that is, the input session – that **StartPenInput** begins. The variable lExtraInfo is the value returned by **GetMessageExtraInfo** called when first processing the message WM_LBUTTONDOWN (see the preceding code fragment). Note that the **StartPenInput** call initiates ink collection, not ink display. The application must take separate steps to begin inking immediately after **StartPenInput** returns.

Inking is the process of displaying a trail of bits behind the tip of the pen as it moves across the surface of the digitizer, simulating the ink dropped by a real pen. If necessary, an application can take on the burden of real-time inking by hooking hardware interrupts with **SetPenHook** and calling the appropriate graphics device interface (GDI) functions to incrementally display ink. However, the Pen API provides a much simpler and more convenient method with the **StartInking** function.

As the **PCMINFO** structure governs **StartPenInput**, the **INKINGINFO** structure determines how **StartInking** operates. To turn on inking with **StartInking**, an application supplies the handle returned by **StartPenInput** and a pointer to the initialized **INKINGINFO** structure, like this:

```
iRet = StartInking( hpcm, LOWORD (lExtraInfo),
                    (LPINKINGINFO) &ink );
```

StartInking offers flexibility in how it displays ink. By modifying values in the **INKINGINFO** structure, an application can change ink color as the pen moves over a specified screen area or it can prevent ink from overwriting a screen object. With the **wFlags** member of **INKINGINFO**, an application can request automatic screen restoration to erase the ink. In this case, Windows replaces the ink trail with the original screen contents overwritten by the ink. This is much faster and simpler than repainting an entire window. Alternatively, an application can prevent ink erasure when pen input ends if, for example, it wants to preserve annotations or other handwritten notes on the screen. The **StartInking** function allows both scenarios.

When **StartPenInput** returns, a stream of WM_PENEVENT messages begins to arrive at the application window procedure. These messages contain submessages that represent current pen activity, such as PE_TERMINATING, PE_PENMOVE, PE_PENDOWN, and PE_PENUP. These submessages represent milestones in the system's ongoing process of collecting data from the pen driver. Each message affords an application the opportunity to gather the raw pen data that has accumulated since the last WM_PENEVENT message.

Windows maintains an internal buffer for data collection, informally named "the ten-second buffer" as a reminder of its limitations. An application should regularly drain the internal buffer by copying data from it at every opportunity afforded by the WM_PENEVENT messages. If it responds to no other event, the application must at least collect data when it receives the PE_BUFFERWARNING submessage, which indicates the internal buffer is more than half full.

To gather the data, an application calls **GetPenInput**. This can be done either in a polling model or in an event-driven model.

In the polling model, the application must repeatedly call **GetPenInput** to get data. It is important for the application to yield periodically; for example, by calling the **PeekMessage** function. A fast loop can potentially process the points before the system can collect more. In this case, successive calls to **GetPenInput** return 0 until the user writes some more. Polling is typically terminated when **GetPenInput**

detects and returns a termination condition specified in **StartPenInput**.

In the event model, the application calls **GetPenInput** in response to each WM_PENEVENT message. The following fragment shows a typical message handler that accumulates ink coordinates in an array of **POINT** structures. The example assumes **StartPenInput** has already been called:

```
POINT       rgPt[MAX_POINTS];           // Array of POINT structures
STROKEINFO  si;                         // Receives pen stroke info
      .
      .
      .
switch (wParam)                         // Process WM_PENEVENT message
{
      case PE_PENDOWN:                   // On any of these events,
      case PE_PENMOVE:                   //   get all points currently
      case PE_PENUP:                     //   in the internal buffer
      case PE_TERMINATING:
      case PE_BUFFERWARNING:
            GetPenInput( hpcm, (LPPOINT) rgPt, NULL, NULL,
                        MAX_POINTS, (LPSTROKEINFO) &si );
            //
            // Latest batch of pen coordinates is now collected
            // into rgPt array.  At this point, the coordinates can be:
            //          (1) Passed to a recognizer (or recognizers)
            //          (2) Passed to a target or control
            //          (3) Placed into an HPENDATA object
            //
                  .
                  .
                  .
            break;

      case PE_TERMINATED:
            // Input session has ended. Do any required
            // clean-up work and display the results.
                  .
                  .
                  .
            break;
```

The example continually calls **GetPenInput** while the pen is in motion until it receives a PE_TERMINATING submessage, indicating the data flow is about to stop. Windows sends a PE_TERMINATING message when it detects one of the termination conditions specified in the **PCMINFO** structure. Typically, the input session ends when the user taps the pen outside a given tablet area or when a specified period of pen inactivity elapses.

An application may need to call **StopPenInput** to stop further data collection. The call to **StopPenInput** is not necessary if the input session ends because of a condition defined in the **PCMINFO** structure. In this case, the system calls **StopPenInput** internally. However, if the application terminates the input session for any other reason, it must call **StopPenInput** explicitly. Unless your application defines all possible termination conditions in a **PCMINFO** structure, it should call **StopPenInput** on detection of a condition that requires termination. Even if the system has already called the function, subsequent calls do no harm.

The preceding description also applies to **StopInking**, provided the application has called **StartInking** to display ink. The system calls **StopInking** automatically if it detects one of the termination conditions

defined in the **PCMINFO** structure; otherwise, the application should call **StopInking** explicitly when required.

Processing the Data

The **GetPenInput** function accumulates the coordinates of the pen stroke in an array of **POINT** structures and places information about the stroke in a **STROKEINFO** structure. This data is "raw" in that it represents a literal history of the pen movement. Some applications will require no more than this. However, further processing of the raw data using other functions of the Pen API usually requires placing the data into an **HPENDATA** or **HRC** object.

The next two chapters examine these objects thoroughly and continue the code fragment outlined previously. Chapter 4, "The Inking Process," describes how an application can alter or manipulate ink data with an **HPENDATA** object. Chapter 5, "The Recognition Process," describes the **HRC** object, which pertains solely to handwriting recognition.

The On-screen Keyboard

The hedit and bedit controls automatically provide user access to the on-screen keyboard. In other situations, an application can display the on-screen keyboard as required by calling the **ShowKeyboard** function.

Besides displaying and hiding the on-screen keyboard, **ShowKeyboard** can also move and minimize the display and select different keyboard types. For a detailed description of the capabilities of **ShowKeyboard**, see Chapter 10, "Pen Application Programming Interface Functions." For other considerations concerning **ShowKeyboard**, see the "Recognition: Use and Misuse" section in Chapter 6.

The Inking Process

This chapter introduces the inking process, in which an application collects and manipulates ink data written by the user. The inking process is a logical next step from the writing process, described in the preceding chapter. In the writing process, the application provides the means for the user to enter ink. In the inking process, the application assembles the ink data, optionally modifies it, and applies it to some task.

The inking process pertains to ink data collected for its own sake rather than immediately passed on to a recognizer for interpretation. Although an application can later submit gathered data to a recognizer, the inking process deals with ink that "stays ink" rather than serving as transitory symbols immediately converted into recognized characters.

The **HPENDATA** data object serves as the major instrument in the inking process. The first part of this chapter describes **HPENDATA** and the various application programming interface (API) functions that serve it. Example code fragments throughout illustrate how to store and manage ink data with the **HPENDATA** services.

An application can refer to the data in an **HPENDATA** object by stroke and point indices, or time intervals. For viewing and manipulating ink data that falls within specified time intervals, the Pen API provides the **HINKSET** object. The last section of this chapter, "The HINKSET Object," examines **HINKSET** and its corresponding API functions.

The HPENDATA Object

An application accesses ink data with **HPENDATA**, which stands for "a handle to pen data." Windows stores the pen data in a block of memory, called the **HPENDATA** object. This data structure is analogous to the other Window "H" data types such as **HDC**, **HCURSOR**, and **HPEN**. **HPENDATA** shares certain similarities with these data types:

- The handle references an internal data structure that resides in memory.
- Windows provides various API functions that operate on the data.
- Developers should ignore the details of the underlying data structure and use the API functions alone to perform the required work.

The remainder of this chapter discusses the **HPENDATA** object and the API functions used to manipulate the data it contains.

Overview of HPENDATA

Windows allocates the **HPENDATA** object with an internal call to the Windows **GlobalAlloc** function. Other Windows data objects are allocated from the USER or GDI heaps, but the large size of an **HPENDATA** object necessitates allocation from the system global heap.

Windows imposes a 64K limit on the size of each **HPENDATA** object. At a report rate of 120 samples per second, at 4 bytes per data point, plus some overhead data structures, minus the time the pen is not in contact with the surface of the tablet, a single **HPENDATA** object can contain the data representing roughly two and one-half minutes of pen activity.

The following section describes the internal structure of an **HPENDATA** memory block. Though not recommended, your application can use this information to read ink data if necessary. However, the internal structure of the **HPENDATA** block may change in future versions of Windows. Therefore, applications should always use standard API functions to read from an **HPENDATA** object.

> **Important** Under no circumstances should an application write directly into an **HPENDATA** block. The Pen API provides functions for modifying ink data safely. Directly changing point data in the block can cause hazardous side effects.

Data Within an HPENDATA Object

Figure 4.1 illustrates the internal structure of an **HPENDATA** object.

{ewc msdncd, EWGraphic, bsd23551 0 /a "SDK_04.BMP"}

Windows stores the pen data in memory in a simple hierarchy. Data points are grouped by strokes in the order in which they are entered. The **HPENDATA** block of memory begins with a descriptive header area. The following sections describe the data points, the stroke headers, and the main header that make up an **HPENDATA** object.

Note that the drawing in Figure 4.1 is not to scale. The data points generally represent a much larger proportion of the memory block than the header components.

Data Points

The data points associated with each stroke are initially tablet coordinates with a resolution of 0.001 inch and an origin at the upper left corner of the tablet. Tablets must report points in this scale regardless of their actual resolution. The Pen API provides functions to scale the points to other metric systems. It is not necessary for the data in an **HPENDATA** object to remain at a resolution of 0.001 inch.

If Windows is running in portrait mode, the tablet still reports coordinates with the upper left corner of the tablet corresponding to the current upper left corner of the display. The developer need not be concerned with the current orientation of the screen. The (0,0) coordinate of the Windows display always corresponds to (0,0) on the tablet.

The **HPENDATA** object can contain additional information supported by the pen device, such as pen pressure, angle, and rotation. The main header section of the **HPENDATA** object specifies how this additional information is stored in the stroke data areas for each data point. Internally, such data, which reflects original equipment manufacturer (OEM) hardware, is stored immediately following the block of coordinates for a stroke. This is called *OEM data*.

Stroke Headers

A *stroke* refers to the data points collected while the pen is in contact with the tablet. These are called *pen-down points*. When the user lifts the pen, the stroke ends. A new stroke begins when the pen next touches the tablet. Some tablets also support *proximity strokes*, which consist of points received when the pen is not in contact with the tablet but near enough for the tablet to sense the pen movement. Such points are called *pen-up points*; a stroke consisting of pen-up points is said to have a *pen-up state*.

As Figure 4.1 shows, a stroke header prefaces each collection of pen coordinates that make up a single stroke. Note that the structure of the stroke header in version 2.0 of the Pen API is different from the stroke header of version 1.0, because, instead of the **STROKEINFO** structure used in version 1.0, the stroke header now consists of a variable-length array . The current **STROKEINFO** structure is still compatible with version 1.0 stroke headers.

Figure 4.1 shows strokes of different sizes. This is because the pen can be in contact with the surface of the tablet for longer or shorter periods of time, resulting in more or fewer points of data. The length of a single stroke is limited only by the 64K maximum size of an **HPENDATA** memory block.

Main Header

A **PENDATAHEADER** structure is the first part of the main header of the **HPENDATA** object. The **PENDATAHEADER** structure, described in Chapter 11, "Pen Application Programming Interface Structures," contains the following information:

- Number of strokes
- Total number of points
- Number of points in longest stroke
- Size in bytes of the memory block
- Bounding rectangle of all pen-down points
- Ink color
- Ink width
- Version

The **wPndts** member of the **PENDATAHEADER** structure describes the state of the data in the **HPENDATA** object. The state of the data reflects whether the data is compressed, includes pen-up points, or includes OEM data. The **wPndts** element is a bitwise-OR combination of the PDTS_ flags described in Chapter 13, "Pen Application Programming Interface Constants."

The next component in the main header is a **PENINFO** structure. The **PENINFO** structure contains information about the tablet device that produced the data. This information includes the tablet width, height, resolution, report rate, proximity capabilities, and barrel-button status. For more information about the **PENINFO** structure, see Chapter 11, "Pen Application Programming Interface Structures."

The **cbOemData** member of the **PENINFO** structure specifies how much (if any) OEM pen data each pen packet contains. The format and order of the extra OEM information are contained in the **rgboempeninfo** member, which is an array of **OEMPENINFO** structures. The **OEMPENINFO** structures describe the order, minimum value, and scale of any OEM pen data the tablet reports along with the coordinate data. Chapter 11 describes the **OEMPENINFO** structure in detail.

HPENDATA Functions

This section introduces the Pen API functions that manipulate ink data in an **HPENDATA** object. The recognition functions described in the next chapter apply to a specific task – the recognition of text. The **HPENDATA** functions, however, are not so easily summed up. They are building blocks for an infinite variety of tasks, according to the requirements and imagination of the developer. For this reason, the best introduction to the **HPENDATA** functions is a simple list of their capabilities.

The following subsections group the functions into six categories and provide a brief description of each function. These subsections serve only as an introduction to the **HPE NDATA** functions. For complete details about the functions, see the appropriate reference sections in Chapter 10, "Pen Application Programming Interface Functions." To see some of the **HPENDATA** functions in use, refer to the PENAPP sample program presented in Chapter 7, "A Sample Pen Application."

The six categories of the **HPENDATA** functions are:

- Creating an **HPENDATA** object
- Displaying ink data
- Scaling ink data
- Examining ink data
- Editing or copying ink data
- Compressing ink data

The order in which functions appear in the following lists reflects either a logical sequence of discussion or, where such criteria do not exist, simple alphabetic ordering. The order does not imply relative importance of the functions.

Creating an HPENDATA Object

The Pen API provides five functions that allocate and free an **HPENDATA** object. These functions are similar to many Windows data types.

Note It is recommended that you use only the functions from version 2.0 of the Pen API. Although API from version 1.0 are included for backward compatibility, it is not guaranteed that they will be supported in future versions of the Pen API.

The functions that allocate and free **HPENDATA** objects are as follows:

Function	Description
CreatePenData	Creates an empty **HPENDATA** object. The application provides the PENINFO structure for the header, the real size of any OEM data stored with each coordinate, and the scale of the coordinates. Superseded by CreatePenDataEx.
CreatePenDataEx	Creates an empty **HPENDATA** object. This function is an enhanced version of CreatePenData that provides an application with greater control over the contents of the **HPENDATA** object.
CreatePenDataHRC	Returns a handle to an **HPENDATA** object copied from an **HRC** object. Since this call creates a new **HPENDATA**, the application should free the object when finished by calling DestroyPenData, described below. The AddPenDataHRC function reverses the process by copying pen data to an **HRC** object. Chapter 5, "The Recognition Process," describes the **HRC** object.
DuplicatePenData	Duplicates an **HPENDATA** object, allowing an application to create clones of existing pen data. Since this call creates a new **HPENDATA**, the application should free the object when finished by calling DestroyPenData, described below.
DestroyPenData	Frees the heap memory occupied by the **HPENDATA** block. If the function returns TRUE, the handle to the object is no longer valid and should be set to NULL.

Displaying Pen Data

The Pen API provides four functions for displaying the pen data contained in an **HPENDATA** object. An additional function, **CreatePenDataRegion**, determines the screen area necessary to display the pen data. This enables an application to determine the screen area affected by a gesture.

The following table describes the API drawing functions:

Function	Description
DrawPenData	Draws pen data in the specified device context using the Windows GDI **Polyline** function. The ink width and color specified in the **PENDATAHEADER** structure have no effect on how **DrawPenData** renders the ink. The rendering of the ink data produced by **DrawPenData** generally does not exactly match the rendering produced by the display driver when the data was first collected. An application that requires an exact replication of the original ink rendering should call the **RedisplayPenData** function.
DrawPenDataEx	Draws pen data in its original color or in a given device context. This function is an enhanced version of **DrawPenData**. Besides basic drawing capabilities, **DrawPenDataEx** can control the speed at which the data is rendered, a process called animation. This function can also draw a selected subset of strokes or the points within a stroke, rather than the entire pen data. **DrawPenDataEx** can display a set of sequential strokes with a single call. Drawing nonsequential strokes – say, the second, fifth, and eighth strokes of the pen data – requires separate calls to **DrawPenDataEx** for each stroke.
DrawPenDataFmt	A macro function that simplifies calls to **DrawPenDataEx** by specifying: • Rendering ink data in original ink attributes and speed (no animation). • Entire data set is drawn (no stroke subsets). • Each stroke is drawn with the color and width specified in the stroke header.
RedisplayPenData	Draws pen data ink that exactly matches the original rendering. **RedisplayPenData** displays pen data with a square GDI pen brush for maximum drawing speed. When displaying wide lines of ink, this optimization can cause the ends of abutting lines to appear "blocky."
CreatePenDataRegion	Returns a region of the screen required to show the contents of an **HPENDATA** object. Another call to the GDI function **GetRgnBox** returns the bounding rectangle that holds the region. (See

also the description of **GetPenDataAttributes**, which can return the bounding rectangle for the entire set of pen data.) The application should call the Windows function **DestroyObject** to free the region when finished.

Scaling Pen Data

The Pen API provides three functions to transform or scale pen data in an **HPENDATA** object. The related functions **TPtoDP** and **DPtoTP** do not operate explicitly on an **HPENDATA** object, but instead convert the resolution of an array of points.

Function	Description
MetricScalePenData	Converts pen coordinates between metric and English standard measurements. Metric units are 0.1 and 0.01 millimeter; English standard units are 0.001 inch. These scaling metrics comply with the mapping modes set in the Windows function **SetMapMode**, described in the Windows Software Development Kit. **MetricScalePenData** can also convert pen data to display resolution. See "Converting Data to Display Resolution" later in this chapter.
OffsetPenData	Offsets the coordinates in an **HPENDATA** object to make them relative to another origin. The function adds or subtracts offset values to or from the coordinate points. The offset values must use the same units as the pen data. Offsetting coordinates does not lose data. The process is completely reversible and does not reduce recognition accuracy.
ResizePenData	Scales ink into arbitrarily sized rectangles. This function exhibits the same weakness as the other scaling functions. It preserves rectangle proportions, but rounding errors prevent the scaling process from being precisely reversible. However, enlarging the ink data generally does not adversely affect recognition accuracy for data later given to a recognizer.

Examining Pen Data

The following functions enable an application to examine, directly modify, or retrieve information from an **HPENDATA** object:

Function	Description
BeginEnumStrokes GetPenDataStroke EndEnumStrokes	These three functions work in tandem. Together, they enable an application to directly read an **HPENDATA** block in memory. Use these functions with caution and only for reading pen data. Do not attempt to write into an **HPENDATA** block. **BeginEnumStrokes** returns a far pointer to the **HPENDATA** memory block within the global heap. **GetPenDataStroke** retrieves pointers to point data within the **HPENDATA**. Although an **HPENDATA** block no longer prefaces strokes with **STROKEINFO** structures, **GetPenDataStroke** can retrieve a **STROKEINFO** structure corresponding to any stroke within the block. When the application has finished with the memory block**,** it must call **EndEnumStrokes**. This unlocks the block in the global heap and invalidates the pointers returned by **GetPenDataStroke**. For this reason, an application must not use the pointers once it has called **EndEnumStrokes**.
GetPenDataInfo	Retrieves summary information from the pen data memory block. It is superseded by the **GetPenDataAttributes** function to some extent.
GetPenDataAttributes	Provides enhanced versions of some of the capabilities of **GetPenDataInfo**. It also provides additional detailed information taken from the **HPENDATA** block. For example, **GetPenDataAttributes** can return • The total number of points in the **HPENDATA** • The total number of strokes • The time the **HPENDATA** was created • The device sampling rate
GetStrokeAttributes SetStrokeAttributes	Retrieve or set information about a given stroke in an **HPENDATA** object. For example, these functions can get or set • The pen state (up or down) for a stroke • The ink color and width • The absolute time the stroke occurred
GetStrokeTableAttributes	Retrieve or set information about all strokes in an **HPENDATA** object that share a given class.

SetStrokeTableAttributes	For all such strokes, these functions can get or set: • The ink color and width. • The user value. • The number of entries in the stroke table.
HitTestPenData	Determines whether a given point lies near the line of a stroke. The function accepts a threshold value that describes a square region around the given point on the tablet or screen. If a stroke in an **HPENDATA** block passes through the square, this function reports a "hit." **HitTestPenData** considers only points with a pen-down state.

Editing or Copying Pen Data

The functions listed below add, delete, or copy coordinate data to and from an **HPENDATA** object.

An empty **HPENDATA** object has the following value for the **wPndts** member of its **PENDATAHEADER** structure:

```
PDTS_NOUPPOINTS | PDTS_NOCOLLINEAR | PDTS_NOEMPTYSTROKES
```

For a full list of PDTS_ values, see Chapter 13, "Pen Application Programming Interface Constants." See Chapter 11 for a description of **PENDATAHEADER**.

Functions in the following list with a name prefix of **Add**, **Insert**, **Extract**, or **Remove** add to or delete from an **HPENDATA** object. These functions check the results of their operation and adjust the value in **wPndts** accordingly.

Function	Description
AddPenDataHRC	Copies the pen data from an **HPENDATA** object to an **HRC** object. This function provides the means for deferred recognition. An application can gather pen data into an **HPENDATA** block, manipulate or store it if desired, then later copy the data to an **HRC** object for recognition.
AddPointsPenData	Appends a set of points, including a **STROKEINFO** structure and any corresponding OEM data, to an **HPENDATA** object.
ExtractPenDataPoints	Copies points and OEM data from an **HPENDATA** block to supplied buffers. The application can convert the points to screen resolution with the **TPtoDP** function to reduce the buffer size. The application can also optionally delete these points from the original pen data.
ExtractPenDataStrokes	Copies selected strokes from an **HPENDATA** block, optionally creating a new **HPENDATA** block containing the copied strokes. **ExtractPenDataStrokes** can optionally delete the original strokes.
GetPointsFromPenData	Retrieves information from an **HPENDATA** object in a way similar to the **GetPenDataStroke** function. However, **GetPointsFromPenData** copies the required data to buffers supplied by the application rather than simply returning pointers to the original data in the global heap. Therefore, the application need not call **BeginEnumStrokes** and/or **EndEnumStrokes** when using **GetPointsFromPenData**.
InsertPenData	Merges two separate **HPENDATA** objects into a single object.
InsertPenDataPoints	Inserts points into the stroke of an existing **HPENDATA** object. **InsertPenDataPoints** automatically updates the stroke and pen data headers.

	This function can adversely affect recognition accuracy for data that must later be recognized.
InsertPenDataStroke	Inserts an entire stroke into an existing **HPENDATA** object. **InsertPenDataStroke** automatically updates the stroke and pen data headers. The ink of the inserted stroke has default color and width. To change these attributes, call **SetStrokeAttributes** after inserting the stroke.
PenDataToBuffer **PenDataFromBuffer**	Copy an **HPENDATA** object as sequential data from and to a buffer. **PenDataFromBuffer** creates and loads an **HPENDATA** object with the data from the sequential buffer created by **PenDataToBuffer**. These functions are used to transfer pen data to and from a file or the clipboard.
RemovePenDataStrokes	Deletes a contiguous set of strokes from an **HPENDATA** object.

Compressing Pen Data

Data compression plays an important role in pen-based computing. The high sampling rates of a pen device, combined with large amounts of input, result in large blocks of ink data. The Pen API offers three methods of compression, each with advantages and disadvantages depending on the intended use of the ink data.

- Removal of redundant or otherwise unwanted data from the data structure. This compression method does not result in loss of recognition accuracy if the compressed data is later recognized.
- *Reversible compression*, also called "lossless" compression. Subsequently decompressing the data produces an **HPENDATA** object identical to the original. Since this compression method loses no information, the data can later be recognized with no loss of accuracy. However, the application cannot copy the compressed data to an **HRC** object; it must first uncompress the data before calling **AddPenInputHRC**.
- *Irreversible compression*, sometimes called "lossy" compression. This method produces the highest degree of compression, but at the cost of lost information. Though the data is still perfectly suitable for display, it cannot be uncompressed and given to a recognizer without a significant loss of recognition accuracy. Irreversible compression is discussed later in the section "Converting Data to Display Resolution."

Compression Functions

Following are the three compression functions provided by the Pen API.

Function	Description
CompactPenData	Provided only for compatibility with version 1.0 of the Pen API and may not be supported in future versions. Use the functions **CompressPenData** and **TrimPenData** instead.
CompressPenData	Primary function used to compress and uncompress pen data.
TrimPenData	Removes selected data from an **HPENDATA** object to reduce the size of the memory block. For example, **TrimPenData** can remove OEM information, timing indexes, pen-up points, and so forth.

Converting Data to Display Resolution

Converting the points in an **HPENDATA** block to display resolution effectively compresses the data because display coordinates are coarser than tablet coordinates and therefore occupy less memory. However, the conversion is irreversible; an application cannot convert the points back to their original tablet resolution. Moreover, converting to display coordinates virtually disallows subsequent recognition of the data because recognizers lose accuracy when dealing with data at coarse screen coordinates.

To compress pen data to screen resolution

1. Call **MetricScalePenData** to convert the ink data from tablet coordinates to display coordinates.
2. Call **TrimPenData** with the TPD_COLLINEAR flag to remove the duplicate and collinear points.

These two steps substantially reduce the number of points in the pen data by removing many high-resolution digitizer points. The following code fragment demonstrates these steps:

```
HPENDATA         hpendata;                    // Handle to HPENDATA object
      .
      .
      .
if (MetricScalePenData( hpendata, PDTS_DISPLAY ))
     iRet = TrimPenData( hpendata, TPD_COLLINEAR );
```

After converting the **HPENDATA** block to display resolution, the application can call **CompressPenData** or **TrimPenData** to compress the points even more. For maximum compression of data intended only for display, use the following instructions instead of the preceding example:

```
if (MetricScalePenData( hpendata, PDTS_DISPLAY ))
     if (TrimPenData( hpendata, TPD_EVERYTHING ) == PDR_OK)
           CompressPenData( hpendata, CMPD_COMPRESS );
```

The HINKSET Object

An *inkset* object consists of time intervals for either individual strokes or a collection of strokes. In turn, the interval of each stroke consists of the times at which the stroke begins and ends. In this way, a pen-based application can refer to a stroke not only by the points it contains but also by the time interval in which the stroke occurs. A rough analogy of this sort of indirect referencing is the way some programming languages allow the use of pointers to indicate data.

Timing information principally serves recognizers. It provides them with an additional characteristic of the raw data that may offer clues for interpretation.

Timing information has other uses, as well. For example, it enables an application to accurately verify a signature by comparing not only the coordinates but the duration of each stroke against a copy of the original signature. This is an effective safeguard against forgery because of the difficulty of simultaneously duplicating both the pattern and duration of the original signature.

An **HINKSET** object can contain up to 5,460 intervals. An interval is expressed as an **INTERVAL** structure, which consists of two **ABSTIME** structures. Each **INTERVAL** structure identifies a stroke's start and stop times in milliseconds. See the appropriate reference sections in Chapter 11 for type definitions of the **ABSTIME** and **INTERVAL** structures.

The HINKSET Functions

The Pen API provides six functions for creating, adding data to, and destroying an **HINKSET** object. The functions automatically ensure intervals within an inkset remain in chronological order.

Function	Description
CreateInkset	Creates an empty inkset into which intervals can be added with the **AddInksetInterval** function.
CreateInksetHRCRESULT	Retrieves the intervals for a specified series of symbols returned by the recognizer.
AddInksetInterval	Adds a single **INTERVAL** structure to an existing **HINKSET** object. Intervals need not be added in any particular order because **AddInksetInterval** automatically sorts the intervals chronologically and merges overlapping intervals.
GetInksetInterval	Copies a series of intervals from an **HINKSET** object to an array of **INTERVAL** structures.
GetInksetIntervalCount	Returns the number of intervals in an **HINKSET** object.
DestroyInkset	Frees the memory occupied by an **HINKSET** object and invalidates its handle.

Timing Information

For uncompressed data, a stroke's interval implies the number of points in the stroke. An application can obtain this number directly from a **STROKEINFO** structure. The following is of academic interest only, illustrating how time intervals correspond to the point data within a stroke.

First, an application gets the device sampling rate with a call to **GetPenDataInfo**. The sampling rate is the number of points the pen device driver sends to Windows during each second of pen activity.

```
HPENDATA        hpndt;
PENINFO         peninfo;
int             nSamplingRate;
     .
     .
     .
if (GetPenDataInfo( hpndt, NULL, (LPPENINFO) &peninfo, 0 ))
    nSamplingRate = peninfo.nSamplingRate;
```

Alternatively, an application can query the pen device driver directly for the sampling rate, as described in "Recognition Functions" in Chapter 8, "Writing a Recognizer."

The number of points in a stroke can now be determined from the start and stop times in the stroke's **INTERVAL** structure:

```
INTERVAL        interval;
int             nSamplingRate, nPoints, nms;
     .
     .
     .
// Compute number of milliseconds in interval
nms = ((interval.atEnd.sec - interval.atBegin.sec) * 1000) +
       (interval.atEnd.ms - interval.atBegin.ms);

// Compute number of points that occurred during interval
nPoints = (nms * nSamplingRate) / 1000;
```

After recognition, an application can determine the time intervals at which recognized symbols were written. Calling **CreateInksetHRCRESULT** creates the inkset for the required intervals:

```
HINKSET     hinkset;            // Allocate the inkset
HRCRESULT   hresult;            // Symbols for guesses go here
     .
     .
     .
// Get symbols that make up the recognizer's best guess
GetResultsHRC( hrc, GRH_ALL, (LPHRCRESULT) &hresult, 1 );

// Get the inkset for symbols 2 through 11 of the guess
hinkset = CreateInksetHRCRESULT( hresult, 2, 10 );
```

The above code fragment creates an inkset containing a maximum of 10 intervals corresponding to the second through eleventh symbols of the recognizer's best guess. The section "Unboxed Recognition" in the next chapter describes the **GetResultsHRC** function in detail. For a description of the internal workings of **CreateInksetHRCRESULT**, see "The Recognition Functions" in Chapter 8, "Writing a Recognizer."

Timing Macros

The PENWIN.H file defines several macro functions designed to deal with timing information. The following reference section briefly describes these macros.

dwDiffAT

dwDiffAT(*at1*, *at2* **)**

Compute time difference in milliseconds between two **ABSTIME** (absolute time) structures *at1* and *at2*. If *at2* is more than *at1*, the result is positive.

Example

```
x = dwDiffAT( at1, at2);           // Get time difference in milliseconds
```

dwDurInterval

dwDurInterval(*lpi* **)**

Calculate the duration in milliseconds of the **INTERVAL** structure *lpi* points to.

Example

```
x = dwDurInterval( lpi );     // Get duration in milliseconds
```

FAbsTimeInInterval

FAbsTimeInInterval(*at*, *lpi* **)**

Test whether the time in the **ABSTIME** structure *at* is contained within the time interval in the **INTERVAL** structure pointed to by *lpi*.

Example

```
if (FAbsTimeInInterval( at, lpi )) // TRUE if at within INTERVAL
```

FIntervalInInterval

FIntervalInInterval(*lpiT*, *lpiS* **)**

Test whether the time interval in the **INTERVAL** structure pointed to by *lpiT* is within the **INTERVAL** structure pointed to by *lpiS*.

Example

```
if (FIntervalInterval( lpiT, lpiS ))     // TRUE if lpiT within lpiS
```

FIntervalXInterval

FIntervalXInterval(*lpiT*, *lpiS* **)**

Test whether the time interval in the **INTERVAL** structure pointed to by *lpiT* overlaps the **INTERVAL** structure pointed to by *lpiS*.

Example

```
if (FIntervalXInterval( lpiT, lpiS ))    // TRUE if lpiT overlaps lpiS
```

FLTAbsTime FLTEAbsTime FEQAbsTime

FLTAbsTime(*at1*, *at2* **)**

FLTEAbsTime(*at1*, *at2* **)**

FEQAbsTime(*at1*, *at2* **)**

Compare two **ABSTIME** structures *at1* and *at2*. These macros return TRUE for the following conditions: less than, less than or equal, and equal.

Example

```
if (FLTAbsTime( at1, at2 ))   // TRUE if at1 < at2
if (FLTEAbsTime( at1, at2 ))  // TRUE if at1 <= at2
if (FEQAbsTime( at1, at2 ))   // TRUE if at1 == at2
```

MakeAbsTime

MakeAbsTime(*lpat*, *sec*, *ms* **)**

Initialize the **ABSTIME** structure pointed to by *lpat* with the values *sec* and *ms*.

Example

```
sec = 5;                          // Time = 5.4 seconds (5 seconds
ms  = 400;                        //   plus 400 milliseconds)
MakeAbsTime( lpat, sec, ms );     // Fill ABSTIME structure
```

The Recognition Process

Recognition is the process of translating pen strokes into characters, symbols, or shapes. An application conducts recognition by passing the raw pen data to special dynamic-link libraries (DLLs) called *recognizers*, each of which may specialize in a particular data type. For example, one recognizer may specialize in English text, another in Greek, and yet another in electronic symbols.

The system default recognizer serves the recognition needs of most applications. Alternatively, applications can load additional recognizers and select which to use for any given input. Regardless of the recognizer, the **DoDefaultPenInput** function can conveniently run the entire recognition process.

An application interfaces with recognizers through a data object called **HRC**, an abbreviation for "handle to a recognition context." This chapter describes the **HRC** and explains how an application uses the **HRC** functions to recognize pen-based handwriting.

The HRC Object

The **HRC** object incorporates a set of functions, called the **HRC** functions, that govern the recognition process. To conduct handwriting recognition, an application creates an **HRC** object, configures its recognition parameters, sends pen data to the object, gives the object time to perform recognition, and eventually gets results from the object.

An **HRC** object provides four different functions, serving as:

- Recognition manager, containing the word lists, templates, alphabets, and **GUIDE** information that aid recognition.
- Data storage, containing the pen coordinates placed into the object with the **AddPenInputHRC** or **AddPenDataHRC** functions. An application can retrieve the entire set of pen data stored in the **HRC** object or only a portion. For example, it can retrieve only the ink associated with a specific letter or word in the recognized text.
- Recognition workhorse, handling the task of recognition through the **ProcessHRC** function.
- Results warehouse, storing recognition results and guesses.

Using the HRC Functions

The following sections describe how an application uses the **HRC** functions to recognize handwritten input as the user writes. The text builds on the steps given in "Starting the Chain of Events" in Chapter 3, "The Writing Process;" refer to that section for more details on each step in the process. The following sections deal with five separate areas of the recognition process:

- Creating the **HRC**
- Configuring the **HRC**
- Processing
- Getting results
- Destroying the **HRC** and other recognition objects

Creating the HRC

Before recognition can occur, the application must create an **HRC** object. **DoDefaultPenInput** does this automatically for the system recognizer, or an application can call the **CreateCompatibleHRC** function to specify another recognizer. **CreateCompatibleHRC** takes two arguments: a handle to an existing **HRC** (if any) that serves as a template for the new **HRC**, and the handle to the recognizer that serves the new **HRC**.

Either or both arguments can be NULL. Giving NULL as the first argument creates a new **HRC** with default settings. The next section, "Configuring the HRC," describes the default parameters, which include the following settings:

- Recognition ends after a brief period of inactivity or when the user taps outside the target window.
- The target window does not use guides.
- The recognizer returns only its best guess without alternative guesses.

Giving NULL as the second argument binds the **HRC** to the system default recognizer. Microsoft Windows sets the supplied file GRECO.DLL as the system default recognizer, specified in the Microsoft Windows 95 system registry. Refer to Appendix A for an explanation of how to change the default to another recognizer.

CreateCompatibleHRC, which is analogous to **CreateCompatibleDC**, copies configuration information from an existing **HRC** to the new **HRC**, which the application can then modify. The following fragment demonstrates how to load a fictitious recognizer called RECOG1.DLL and bind it to a new **HRC** patterned after an existing **HRC** called hrcTemplate:

```
HRC     hrc1;                       // Handle to new HRC
HREC    hrec1;                      // Module handle of recognizer
     .
     .
     .
hrec1 = InstallRecognizer( "RECOG1.DLL" );
if (hrec1)
     hrc1 = CreateCompatibleHRC( hrcTemplate, hrec1 );
```

Each **HRC** can access only one recognizer and the binding lasts the life of the **HRC**. To use multiple recognizers, an application must create multiple **HRC** objects, binding each to a different recognizer. The Pen API does not provide a means for changing the recognizer associated with an **HRC**.

As shown in the example above, an application must call **InstallRecognizer** to load any other recognizer it will use. The exception is the system default recognizer, which is already installed when the system starts up. An application should not install the system recognizer with **InstallRecognizer**. Doing so only creates an unnecessary module handle.

To preserve system resources, an application must free all handles obtained from **InstallRecognizer** with separate calls to **UninstallRecognizer**. Unlike other DLLs, a recognizer belongs to the system instead of the application. Windows does not unload the recognizer from memory until every client has called **UninstallRecognizer**.

Once it receives a valid **HRC** handle, the application can begin configuring the **HRC** to perform handwriting recognition.

Configuring the HRC

Before passing data to the **HRC**, the application must ensure the **HRC** is properly configured to perform recognition. The configuration information in an **HRC** provides a context to guide the recognition process. For example, if the application expects only numerical input, it can configure the **HRC** accordingly. This prevents the recognizer from mistakenly confusing the numeral "0" for the letter "O."

A new **HRC** shares the same configuration as the template **HRC** given when calling **CreateCompatibleHRC**. If the application does not provide a template for **CreateCompatibleHRC**, the new **HRC** receives a default configuration.

The following paragraphs describe the various configurations an application can set for an **HRC** object.

Alphabet

The **SetAlphabetHRC** function specifies a set of symbols the **HRC** should expect in the input. (A similar function, **SetBoxAlphabetHRC**, provides the same service for a group of boxes when the **HRC** uses guides.) For example, the application can constrain recognition to numerals and uppercase letters, as shown here:

```
iRet = SetAlphabetHRC( hrc1, ALC_NUMERIC | ALC_UCALPHA, NULL );
```

For more details about **SetAlphabetHRC** and **SetBoxAlphabetHRC**, see Chapter 10, "Pen Application Programming Interface Functions."

The first argument of **SetAlphabetHRC** is the **HRC** handle returned by **CreateCompatibleHRC**. The second argument is a bitwise-OR value formed by the desired combination of ALC_ constants, some of which are listed here:

Alphabet constant	Description
ALC_DEFAULT	Default alphabet value for recognizer. If recognizer can serve as a system recognizer, its default alphabet must include at least the ALC_SYSMINIMUM set. The Pen API does not specify a default for nonsystem recognizers.
ALC_LCALPHA	Lowercase letters: a-z.
ALC_UCALPHA	Uppercase letters: A-Z.
ALC_NUMERIC	Numerals: 0-9.
ALC_ALPHANUMERIC	Combines ALC_LCALPHA, ALC_UCALPHA, and ALC_NUMERIC.
ALC_PUNC	Punctuation: !-;`"?()&.,;\.
ALC_MATH	Math symbols: %^*()-+={}<>,/.
ALC_MONETARY	Monetary symbols: ,.$ (or as determined by language).
ALC_OTHER	Other special characters: @#|_~[].
ALC_ASCII	Seven-bit characters ASCII #20 to ASCII #127.
ALC_WHITE	White space such as tabs and newline and space characters.
ALC_NONPRINT	TAB, ENTER, and CTRL keys.
ALC_SYSMINIMUM	Combines ALC_ALPHANUMERIC, ALC_PUNC, ALC_WHITE, and ALC_GESTURE.

If an application does not specify alphabet configuration either through an existing **HRC** model or by calling **SetAlphabetHRC** or **SetBoxAlphabetHRC**, Windows assumes ALC_SYSMINIMUM as the default alphabet configuration. For a complete list of ALC_ values, see Chapter 13, "Pen Application Programming Interface Constants."

Gesture

A default **HRC** enables all gestures. An application can disable certain gestures by calling the **EnableGestureSetHRC** function to change the gesture configuration for the **HRC**. The following example disables the gestures for cut, copy, and paste while enabling all other gestures:

```
iRet = EnableGestureSetHRC( hrc1, GST_ALL ^ GST_CLIP, TRUE );
```

For more information about **EnableGestureSetHRC**, see the reference section in Chapter 10, "Pen Application Programming Interface Functions."

The first argument of **EnableGestureSetHRC** is the **HRC** handle returned by **CreateComptibleHRC**. The second argument is a bitwise-OR value formed by the desired combination of GST_ constants, listed here:

Gesture constant	Description
GST_CLIP	Cut, copy, and paste.
GST_WHITE	Space, tab, and newline.
GST_SEL	Lasso selection.
GST_EDIT	Insert, correct, and undo.
GST_SYS	Combines all the above.
GST_CIRCLELO	Lowercase circle gestures.
GST_CIRCLEUP	Uppercase circle gestures.
GST_CIRCLE	Combines GST_CIRCLELO and GST_CIRCLEUP.
GST_ALL	Combines all the above.

Word List

An application can select a word list from any number of lists to attach to an **HRC**. A word list, referenced by a handle, consists of words the recognizer should consider when translating a handwritten word or phrase. For example, Figure 5.1 shows a case in which the recognizer must decide between the valid interpretations of "boy" and "looy".

{ewc msdncd, EWGraphic, bsd23552 0 /a "SDK_05.BMP"}

By consulting a word list that contains the entry "boy" but not "looy", the recognizer can select the first choice with more confidence.

An application must call the **CreateHWL** function to create a word list. The function accepts a pointer to a word list already in memory. Alternatively, an application can fill the word list from an existing file through the **ReadHWL** function. The **SetWordlistHRC** function attaches the word list to a particular **HRC**, and **DestroyHWL** returns the memory occupied by the list to the operating system.

The function **SetWordlistCoercionHRC** allows an application to establish the word list's influence over the recognizer's interpretations. The function accepts the following values to set the degree of coercion:

Coercion value	Description
SCH_ADVISE	The recognizer should use the word list only for hints; results are not strongly coerced to match the word list. This is the default coercion value.
SCH_FORCE	If the recognizer does not find an exact match in the word list, it should return the best fit. For example, if the recognizer interprets a handwritten word as "swoden," it will return "Sweden," given a word list of country names.
SCH_NONE	Cancels any coercion currently in effect.

The following code fragment illustrates all these steps by reading a list of country names from the fictitious file COUNTRY.LST and attaching the list to the **HRC** identified by the handle `hrc1`. It calls **SetWordlistCoercionHRC** to force the recognizer to return only names found in the COUNTRY.LST file. The code assumes `vhwl` is a global variable, visible in all parts of the program.

```
HWL             vhwl;                       // Handle to word list
HFILE      hfile;                           // File handle
OFSTRUCT        OFstruct;                   // Receives info about open
file
int             iRet;                       // Return code
     .
     .
     .
// In intialization procedure, open and read the word list
hfile = OpenFile( "COUNTRY.LST", (LPOFSTRUCT) &OFstruct, OF_READ );
if (hfile != HFILE_ERROR)
{
     vhwl  = CreateHWL( hrec1, NULL, WLT_EMPTY );    // Create empty list
     iRet  = ReadHWL( vhwl, hfile );                      // Read list from
file
}
     .
     .
```

```
      .
// After creating hrc1, attach word list vhwl to it
SetWordlistHRC( hrc1, vhwl );                        // Attach list to hrc1
SetWordlistCoercionHRC( hrc1, SCH_FORCE );           // Establish coercion
      .
      .
      .
//Before terminating, destroy word list
DestroyHWL( vhwl );
```

Note that an application must first set the word list with **SetWordlistHRC** before calling **SetWordlistCoercionHRC**.

Guide

Guides are visual cues such as lines or boxes in a bedit control. Guide configuration informs the recognizer of the types and locations of guides displayed for the user. With this information, the recognizer can confidently determine which pen strokes constitute a single character or shape. For an illustration of box guides, see the "The bedit Control" section in Chapter 3, "The Writing Process."

To establish guide configuration, an application calls the **SetGuideHRC** function, providing a pointer to a **GUIDE** structure:

```
GUIDE      guide;
      .
      .
      .
iRet = SetGuideHRC( hrc1, (LPGUIDE) &guide, 0 );
```

Number of Recognition Guesses

The application can specify the maximum number of guesses the recognizer should provide in its results. This allows the application to prevent the recognizer from generating more alternative guesses than required.

Set the maximum number of guesses with the **SetMaxResultsHRC** function. The following example code tells the recognizer to provide its five best guesses:

```
iRet = SetMaxResultsHRC( hrc1, 5 );
```

The default number of maximum recognition results is 1.

Processing

Once the **HRC** object has been properly created and configured, it can take on its role of recognition agent. To fulfill this role, the **HRC** requires:

- The data generated by the pen movement.
- Sufficient central processing unit (CPU) time to execute the recognition algorithms and generate results.

The following sections describe how an application supplies these two requirements to the **HRC** object.

Adding Data to an HRC Object

An application provides pen data to the **HRC** through one of two API functions: **AddPenInputHRC** or **AddPenDataHRC**.

AddPenInputHRC operates at intervals as pen data is collected, in the same manner as **GetPenInput**. An application must call **AddPenInputHRC** only after it has called **GetPenInput**. **AddPenInputHRC** provides the pen coordinates and original equipment manufacturer (OEM) data to the recognizer bound to the **HRC** object.

AddPenDataHRC also provides pen data to the recognizer, but is designed to operate after all the data is collected. An application can thus collect pen data without realtime recognition, store the data in an **HPENDATA** structure, and call **AddPenDataHRC** to recognize the data any time thereafter.

Allocating Processing Time

After it has supplied the raw pen data to an **HRC** object, the application then allocates processing time for recognition by calling **ProcessHRC**. To accommodate applications with time-critical communications requirements or other CPU-intensive activities, **ProcessHRC** takes a time-out value (in milliseconds) as its second argument. If the time-out period elapses before **ProcessHRC** finishes processing, the function returns an HRCR_INCOMPLETE value. In this way, an application can repeatedly allocate small slices of time until the recognizer finishes its work.

PENWIN.H defines three time-out codes that an application can use when calling **ProcessHRC**. The following table describes the time-out codes.

Time-out code	Description
PH_MIN	Allocates the smallest possible period of time to the recognizer, approximately 50 milliseconds.
PH_DEFAULT	Allocates a moderate amount of time to the recognizer, approximately 200 milliseconds.
PH_MAX	Grants the recognizer as much time as it requires to complete the recognition.

The following line allocates the default time-out period to the recognizer in the **HRC** identified by the handle hrc1:

```
iRet = ProcessHRC( hrc1, PH_DEFAULT );
```

Typically, **AddPenInputHRC** and **ProcessHRC** work together in a loop or in repeated response to a PE_ message as the user writes. One function continually retrieves the latest pen data while the other processes that data. When the input session terminates, an application should call **ProcessHRC** with PH_MAX to finalize the recognition.

See the reference section for **ProcessHRC** in Chapter 10 for additional information about this function. When **ProcessHRC** returns, the application can retrieve results from the **HRC** object, as described in the next section.

Getting Results

The Pen API provides three methods for an application to get recognition results. The first two methods are functions – **GetBoxResultsHRC** and **GetResultsHRC** – that retrieve the results from an **HRC** object. The **GetBoxResultsHRC** function assumes the application has provided a **GUIDE** structure to the **HRC**. If the application has not specified a **GUIDE** structure, it must call **GetResultsHRC** to retrieve an **HRCRESULT** object for each alternative guess.

The third method applies only to version 1.0 recognition functions. It retrieves recognition results for both boxed and unboxed input from an **RCRESULT** structure. With this method, an application must dissect the **RCRESULT** structure to get information that **GetBoxResultsHRC** and **GetResultsHRC** provide automatically.

The rest of this section gives examples for each of the three methods.

Boxed Recognition

GetBoxResultsHRC retrieves boxed recognition results on a box-by-box basis. The function fills an array of **BOXRESULTS** structures with the results and alternatives for a set of boxed character positions in the recognized text. Each of the **BOXRESULTS** structure in the array contains one or more recognition guesses for a single box.

For example, consider the case in which the user has written a seven-letter word before requesting recognition – say, by tapping an OK button. The application can retrieve results individually for each box by calling **GetBoxResultsHRC** in a loop, or provide an array of at least seven **BOXRESULTS** structures and receive all seven recognition results with a single call to **GetBoxResultsHRC**.

The following example code retrieves recognition results two boxes at a time. It requests only the first alternative for each box, which represents the recognizer's best guess about the character in the box:

```
#define                NBOX  2                  // Number of boxes to get
#define                NALT  1                  // Only one alternative per
box

BOXRESULTS       box[NBOX];              // Array of BOXRESULTS structures
UINT             iSyv = 0;               // Index of symbol values
int                    cBox;                    // Number of results returned
     .
     .
     .
do {
     cBox = GetBoxResultsHRC( hrc1, NALT, iSyv, NBOX,
                                    (LPBOXRESULTS)&box, FALSE );
          .                                                        // Do error
checking
          .                                                        // Read
results from
          .                                                        //   box[]
array
     iSyv += (UINT) cBox;                                  // Increment
index
} while (cBox == NBOX)                                 // Loop for next boxes
```

By requesting only a single alternative for a small number of boxes, the preceding example can allocate the array on the stack. However, the **BOXRESULTS** structure contains a variable-length array of type **SYV** for additional alternative characters. For a value of NALT greater than 1, the application must allocate extra space for the alternatives in each **BOXRESULTS**, as the following line demonstrates:

```
HGLOBAL    hMem = GlobalAlloc( GHND,
                          NBOX*( sizeof(BOXRESULTS) + (NALT-
1)*sizeof(SYV)) );
```

The example provided in the **GetBoxResultsHRC** reference section in Chapter 10 further illustrates how to use this function.

When **GetBoxResultsHRC** returns, the application can walk the **BOXRESULTS** array and display the information appropriately. The boxed edit control described in the "The bedit Control" section in Chapter 3 uses **GetBoxResultsHRC** to perform recognition and to generate alternative results.

Boxed writing does not constrain an application to call **GetBoxResultsHRC** for recognition results. An application can also call **GetResultsHRC**, described next, even if the **HRC** is configured for box guides.

Unboxed Recognition

For recognition of unboxed handwriting – that is, writing without visual guides as specified by a **GUIDE** structure, an application must call the **GetResultsHRC** function. This function fills an array of **HRCRESULT** objects, each containing a separate guess by the recognizer. The number of **HRCRESULT** objects in the array is always less than or equal to the maximum number of guesses requested through the **SetMaxResultsHRC** function.

An example will clarify this. Assume an application contains the following instructions:

```
#define          MAX_GUESS   5          // Maximum number of guesses
allowed
      .
      .
      .
SetMaxResultsHRC( hrc1, MAX_GUESS );
```

The user next writes a word that the recognizer associated with hrc1 guesses to be, in descending order of probability, either "clear," "dear," "clean," "dean," or "deer." Though it might have generated even more guesses, the recognizer is constrained to stop after its fifth guess by the earlier call to **SetMaxResultsHRC**. In this case, a subsequent call to **GetResultsHRC** fills an array of up to five **HRCRESULT** objects, the first **HRCRESULT** containing the word "clear," the second the word "dear," and so forth.

An **HRCRESULT** object does not contain a normal ASCII string representation of a guess. This is not possible since a guess might be made up of a gesture, shape, or other entity that has no ASCII equivalent. Instead, an **HRCRESULT** contains a string of *symbol values*, which are 32-bit numbers type-defined as **SYV**.

Symbol values can represent geometric shapes, gestures, letters of the alphabet, Japanese Kanji characters, musical notes, electronic symbols, or any other symbols defined by the recognizer. The Pen API provides the function **SymbolToCharacter** to convert the null-terminated symbol string in **HRCRESULT** to an ASCII string.

The following code continues the example above, illustrating how to retrieve and display the five guesses returned by the recognizer:

```
#define     MAX_GUESS   5       // Maximum number of guesses allowed
#define     MAX_CHAR       50      // Maximum number of characters in
                                //   a single guess
HRCRESULT   result[MAX_GUESS];  // Array for recognition result objects
int         cGuess;
      .
      .
      .
SetMaxResultsHRC( hrc1, MAX_GUESS );
      .
      .
      .
//
// Get all (non-gesture) guesses available, and if no errors,
// convert to ASCII strings and display them.  Note in our
// example the following call returns the value 5 to cGuess.
//
cGuess = GetResultsHRC( hrc1, GRH_NONGESTURE,
                        (LPHRCRESULT) &result, MAX_GUESS );
```

```
if (cGuess > 0)
{
     int     i, cSyv, cChar;
     char    szText[MAX_CHAR];           // Buffer for converted text
     SYV     rgSyv[MAX_CHAR];            // Buffer for symbol string

     //
     // Loop cGuess (5) times, retrieving each time a symbol string
     // representing a different guess.  Convert the symbol string
     // to a normal ASCII string and display it.
     //
     // In our example, the five iterations of this loop display the
     // words "clear," "dear," "clean," "dean," and "deer."
     //
     for (i = 0; i < cGuess; i++)
     {
           cSyv = GetSymbolsHRCRESULT( result[i], 0, rgSyv, MAX_CHAR );
           if (cSyv > 0)
           {
             SymbolToCharacter( (LPSYV) rgSyv, cSyv,
                                (LPSTR) szText, (LPINT) &cChar );
                        .
                        .                                    // After
converting to ASCII
                        .                                    //   string,
display text
             TextOut( hdc, nX, nY, (LPSTR) szText, cChar );
                        .
                        .
                        .
           }
        DestroyHRCRESULT( result[i] );  // When finished with results,
        result[i] = NULL;               //   destroy HRCRESULT objects
     }
}
```

For another example of how to use the **GetResultsHRC** function, see "Step 9: PE_RESULT Message" in Chapter 2. If an application must retrieve an unknown number of symbol values from the recognizer, it should follow these three steps:

1. Call **GetSymbolCountHRCRESULT** to determine the number of symbol values the recognizer can return.
2. Allocate a sufficient buffer for the values.
3. Call **GetSymbolsHRCRESULT**. to copy the symbol values to the buffer.

Getting Results from the RCRESULT Structure

Note The **RCRESULT** structure is supported only for backward compatibility. It may not exist in future versions of the Pen API. Applications should obtain recognition results through the API functions described in this chapter, rather than from an **RCRESULT** structure.

The **RCRESULT** structure applies only when an application calls either of the version 1.0 recognition functions, **Recognize** or **RecognizeData**. In this case, the system sends a WM_RCRESULT message to the application. The *wParam* of this message contains a REC_ submessage that indicates why recognition ended. The *lParam* of WM_RCRESULT points to an **RCRESULT** structure, which contains all the results of the recognition.

An application can retrieve from the **RCRESULT** structure all the recognizer's guesses by walking through the list, called the symbol graph, contained in the **RCRESULT** structure.

The **RCRESULT** structure identifies the recognizer's "best guess," which is the guess in which the recognizer places the most confidence. With this information, an application can conveniently retrieve an ASCII string of the best guess by calling **SymbolToCharacter**:

```
char szBestGuess[MAX_CHAR];        // ASCII string of best guess
     .
     .
     .
switch (wMsg)
{
     case WM_RCRESULT:
          SymbolToCharacter(
                        (LPSYV) ((LPRCRESULT)lparam)->lpsyv,      // Symbol
string
                        MAX_CHAR,                                       //
Maximum length
                        (LPSTR) szBestGuess,                            //
Buffer for ASCII
                        NULL );
     // Don't need count
```

Compare the above call to **SymbolToCharacter** with the previous example. Here, the second argument represents a maximum, rather than the actual length of the symbol string, which is the value

```
(int) ((LPRCRESULT)lparam)->cSyv
```

By specifying the length of the buffer that receives the ASCII text, the second argument sets a cap on the number of symbols **SymbolToCharacter** will convert. This prevents the function from overflowing the szBestGuess buffer if the length of the symbol string happens to be larger than MAX_CHAR. **SymbolToCharacter** returns when it encounters SYV_NULL at the end of the symbol string or when it converts MAX_CHAR symbols, whichever occurs first.

An application that calls **Recognize** or **RecognizeData** must be prepared to receive WM_RCRESULT messages before calling either function. This is because the recognizer dispatches all WM_RCRESULT messages associated with a particular recognition event before **Recognize** or **RecognizeData** returns.

Version 2.0 of the Pen API provides, through function calls, all the information contained in an **RCRESULT** structure. An application need not examine the structure at all. **RCRESULT** is a product of recognition, and is therefore of more interest to the recognizer developer than the application developer.

Consequently, it is described in more detail in Chapter 8, "Writing a Recognizer."

Destroying the HRC

The useful life of an **HRC** object usually expires when the recognizer returns results at the end of an input session. The next input session requires the creation of a new **HRC**. When finished with an **HRC** object, an application should destroy the object in two steps:

1. Call **DestroyHRC** to free the occupied memory.
2. Set the **HRC** handle to NULL.

For example:

```
     HRC          vhrc = NULL;                          // Set to NULL until
HRC is created
            .
            .                                          // Create and use the
HRC
            .
     DestroyHRC( vhrc );                         // Destroy the HRC when
finished
     vhrc = NULL;                                       // Handle is now
invalid
```

When **DestroyHRC** returns, the handle value remains unchanged though no longer valid. The second step above prevents frustrating bugs arising from the inadvertent use of an invalid **HRC** handle.

This same advice applies to the other recognition objects, **HRCRESULT** and **HWL**. After calling **DestroyHRCRESULT** or **DestroyHWL**, always set the invalid handle to NULL.

Design Considerations

The developer of pen-based applications should bear in mind the unusual qualities of a pen interface. Input through a pen device provides unique advantages, yet at the same time carries severe limitations. The best applications will seek to profit from one quality while minimizing the effects of the other.

This chapter discusses some of the characteristics of an intelligent and responsive pen-based application. It offers tips, ideas, and a few warnings. The advice is based on experience and the results of usability studies conducted by Microsoft.

Basic Principles

Consider the following basic principles when designing the user interface of a pen-based application. Though not intended to constrain the developer's creativity, these principles can help ensure that the resulting application appeals to its users.

- Keep it simple.
- Use familiar models.
- Show feedback for user.
- Make it fast.
- Make it fun.
- Make exploration safe.
- Let the user maintain control.

The following sections explore each of these guidelines.

Keep It Simple

The developer should value simplicity over power when designing a pen-based application. Simplicity is not only a characteristic of good interface design, it hastens the user's acceptance of a type of input paradigm apt to be new and unfamiliar.

The same principles for writing a standard Windows-based application apply equally to pen-based programs:

- Limit features and options to reduce the number of choices a user must make. When adapting an existing application to run on a pen-based computer, remember the so-called "80/20" rule: 80 percent of an application's value is typically provided by only 20 percent of its features.
- Keep the interface clear, consistent, uncomplicated, and predictable. The relationship between what a user does and how the application responds should be logical and consistent. Keeping the interface consistent and predictable reduces the amount of information the user must remember in order to use an application.
- Make possible actions and results visible to the user. Enable the user to work directly with objects without resorting to abstractions. The user wants to "send mail" or "find a note," not "open an application" or "search for data."
- Use constraints to prevent the user from choosing inappropriate actions and provide default choices whenever appropriate. Constraints encourage the user to make appropriate decisions by limiting unlikely choices. For example, a button enabling a user to save or pause a game should not be visible until play has started.

Use Familiar Models

Familiar conceptual models are powerful aids in user-interface design. A conceptual model enables users to apply knowledge gained from experience toward understanding the structure and use of the application. For example, an Address application modeled after a typical paper-based address book would allow users to apply their understanding of address books to the new application.

Use Feedback

The user should receive immediate and tangible feedback during interaction with an application. Appropriate feedback includes acknowledging a request, pointing out an error, or tracking the progress of an operation. Although auditory feedback can be useful for attracting a user's attention, it should be used sparingly in a pen-based application for the following reasons:

- Many users find beeps annoying.
- Pen-based computers will be used more and more frequently in conference rooms and other group areas where beeping from a machine will not be welcome.
- Auditory messages disappear without a trace. If the user is momentarily away or distracted, the auditory signal has failed to do its job.
- If the user can turn the warning sound off, sound is not a reliable source of feedback.
- An audible notification is not useful for users who are deaf or hard of hearing.

Make It Fast

A simple and responsive interface is more appealing than an attractive yet sluggish interface. An application should always be ready for user input and prepared to offer immediate feedback. Ideally, results should quickly follow the user's actions.

Where immediate results are not possible, run lengthy operations in separate threads if practical. This technique has the advantage of at least simulating speed by returning control quickly to the user.

Make It Fun

Users will look for applications that have simple, creative interfaces that are fun to use when deciding what to keep on their portable pen-based machines. Paying special attention to the visual appeal of an interface pays off in gaining user acceptance. The most powerful interfaces are those that combine aesthetics with functionality.

Make Exploration Safe

People like to explore applications and learn by trial and error. Such selfmotivated learning is extremely effective, but users might not always be aware of potential dangers. Even with the best-designed interface, users make mistakes – such as accidentally tapping the wrong object or data, or making a wrong decision about which data to select. The interface should accommodate user exploration by:

- Softening any penalty caused by mistake.
- Minimizing the opportunities for errors.
- Handling user errors gracefully, without implying the user is at fault.
- Allowing easy undo and undelete.
- Keeping separate training databases to accommodate guest users.

Let the User Maintain Control

People want to feel in control of an application. A well-designed, responsive user interface contributes much toward the user's perception of being in control. The following list gives design suggestions for achieving this:

- Enable the user to interrupt long operations.
- Discard meaningless user input during long operations. While waiting, users might randomly tap on the pen tablet or display. To dispose of spurious input in a discreet manner, enable your application to distinguish between meaningful and unintentional data.
- Allow the user to specify desired default settings.

Recognition: Use and Misuse

Recognition is often the deciding factor in how people react to pen-based computing. It is, unfortunately, an inexact science and always will be. The best applications will seek to minimize the potential for error introduced by recognition by restricting as much as possible the amount of recognition necessary. The following may give you some ideas.

Selecting Is Better Than Writing

As much as possible, let the user select rather than write. For example, take advantage of spin-box and list-box controls that don't require written input. When prompting for a date, present the user with a simple calendar on which he or she can pick a date by tapping with the pen.

Keep track of previous entries and allow the user to select one from a list rather than having to rewrite it. For example, when prompting for a name, consider using a combo-box control to provide access to a list of the previous 10 or so names the user has last entered. This confines the potential for recognition errors to the first time the name is written.

When prompting for a filename, provide an option for browsing through directory lists, allowing the user to select a file and path by tapping the filename with the pen.

bedit Is Better Than hedit

People prefer to write in the relatively unrestricted space of an hedit control, but the bedit control offers better recognition accuracy. The comb and box guides of a bedit also serve as discreet prompts, informing the user that the application awaits written input.

If saving screen space is important, consider using a lens instead of a bedit. Always create a single-line edit control with ES_AUTOHSCROLL so it shows a lens button.

Real Time Is Better Than Deferred Time

Deferred recognition offers the seductive advantages of speed and instantaneous response. By collecting ink without pausing to recognize it, an application can easily keep up with rapid pen movements. The input can be recognized later when requested by the user, or perhaps during periods of user inactivity.

However, Microsoft tests have demonstrated that the accuracy of deferred recognition often compares unfavorably to real-time recognition. This has nothing to do with the recognizers, since they apply the same processing procedures to the data regardless of when the ink is collected. The discrepancy arises from the fact that people tend to write more carelessly if not continually informed about the recognizer's success rate.

By seeing the recognition results as they write, users naturally adapt their writing speed and style to assure the greatest recognition accuracy. Although in theory the user should train the recognizer, to a certain extent the reverse undeniably occurs.

Make Corrections Easy

Users don't mind recognition errors as much as they mind the effort required to correct the errors. A good pen model focuses on making corrections as easy and fast as possible.

An error in recognition should never have unpleasant consequences. For example, misrecognizing an undo gesture wastes the user's time. When in doubt, prompt for confirmation and make confirmation easy − say, with an extra-large OK button. The extra step will annoy the user less than having to recover from the error.

Provide Easy Access to the On-screen Keyboard

The on-screen keyboard serves well for short input, especially for correcting erroneous recognition results. An application should ensure that the on-screen keyboard is easily accessible to the user, yet remains unobtrusive. Microsoft usability studies have shown that users prefer writing to tapping on the on-screen keyboard, even though the latter is often faster because it avoids recognition errors. Consider making the on-screen keyboard more or less noticeable depending on whether your application runs on a desktop system with a real keyboard or on a handheld unit with no physical keyboard.

Other Considerations

The developer of pen-based applications should consider other facets of pen computing besides recognition. This section lists a few ideas.

Don't Rely on Gestures

In a well-designed pen application, all operations are possible without gestures. The application may support gestures as shortcuts, but should not sacrifice common operations for the sake of the gesture.

Gestures also tend to remain a hidden (or "nondiscoverable") functionality, which the novice user often does not guess at. Gestures should facilitate the experienced user without hampering the uninitiated.

Action handles provide the same benefits as gestures. Moreover, they are more discoverable and reliable, since they do not require recognition.

Every pen application should, at minimum, support the cut and lasso gestures. Anything else is at the discretion of the developer. Incidentally, usability tests have found that a common gesture among novice users is to scribble over an entry to erase or undo it. An intelligent application should respond to such unknown gestures and display a polite inquiry, listing possible alternative actions that the user can select by tapping.

Provide Ample Target Space

The pen often proves an inaccurate pointing device. The well-behaved pen application allows ample margin for pointing errors from the user, who can easily miss a small button or other target. Consider the following suggestions to make your pen-based application friendly to the user with poor aim:

- Create targets as large as practical.
- Space toolbar buttons so that they have gaps between them.
- Avoid crowding dialog boxes with controls placed near one another.
- Pen-down events falling within a few pixels of a button should be treated as a press of that button. Always increase the effective size of a control by sending the HE_SETINFLATE submessage, as described in the "HE_SETINFLATE Submessage" section in Chapter 3, "The Writing Process."
- Compensate for the pixel sizes of different displays. Use **GetDeviceCaps** to determine sizes and maintain uniform dimensions for on-screen targets.

Use Position Clues

A pen-based application should consider position clues in determining the user's intentions. The following offers a few examples of inferring the user's desires from the position of the ink:

- A gesture drawn over part of a selection should operate on the entire selection. In the same vein, a gesture or lasso that intersects more than a single letter of a word is probably meant for the entire word.
- Writing text on a line below existing text serves as a good indication the user intends the new text to go on a new line. In this case, an application can insert a newline character automatically.
- Text written over an insertion point should be inserted at the insertion point.

Conserve Power

A pen-based environment will often be found on small notebook- or handheld-size computers. Users will appreciate pen-based applications that extend battery life by conserving power. Here are a few power-saving tips:

- Avoid "disk hits" as much as possible. Hard disks on small systems often turn off after a period of inactivity and powering them up again significantly affects battery life. An application should avoid unnecessarily accessing the disk, since doing so may force the system to repower the drive.
- Keep code and data files small to minimize the disk swapping Windows must do to clear memory. Restrict the number of dynamic-link library (DLL) files your application requires and consider loading the DLLs early. In this way, Windows reads the DLL files immediately after having loaded the application itself, while the disk is still in motion. Linking to a DLL's import library ensures that the DLL is installed at the same time as the application.

  However, this advice applies only to small DLL files and DLLs that the application will most likely use at some point. Large DLL files that stand a good chance of not being required by the application should be loaded explicitly only when needed by calling the **InstallRecognizer** function or the **LoadLibrary** function. Although this risks powering up a dormant disk drive, it also prevents unneeded objects from occupying memory.
- Prefer visual to auditory signals. Besides the negative reasons cited in the "Use Feedback" section, beeps from the speaker also waste battery power.
- Reduce video power drain by making background colors black.

Guidelines for Applications

The following presents some recommended approaches for different types of applications, based on the guidelines presented above. While by no means exhaustive, the material may give you some ideas for various types of applications.

Annotation

Many different types of software can benefit from the unique advantages of pen-based *annotation*. A user annotates by writing with a pen on top of an existing document, as though on an overlaying transparent sheet. This allows adding to the document free-form writing such as notes, diagrams, review comments, questions, and so forth.

Unless evident by the context, an application should prompt the user to identify a position in the document to attach the annotation. This prevents the annotation from drifting from its intended location if the underlying document is changed.

Often, annotated text remains unrecognized, captured as ink data. Reduce such text to display resolution to minimize the file space it occupies. For more information on how to achieve maximum compression of handwritten text, see "Converting Data to Display Resolution" in Chapter 4.

Word Processor

Although the pen does not serve well to create a word-processor document, it can do so for small editing tasks on existing documents, such as for cut-and-paste operations, formatting changes, rewriting small amounts of text, and navigation (scrolling). Thus, the pen in a word-processing application should behave as a pointing device most of the time. The user should be able to select text by dragging the pen and the selection should include an action handle. Double-tapping should display a writing window, as should an "edit" command on a selection action handle. Consider also providing an insertion-point action handle, including an "insert text" command.

When the user creates a new blank document, the application should automatically display an editing window because the user clearly intends to enter new text. For existing documents, the application should provide a means for annotating the text, either by inserting scribble notes (like Post-it™ notes) or by inking directly on top of the text.

Spreadsheet

Within the spreadsheet area, the pen should default to a pointer. If the application allows in-cell editing, double-tapping the cell should open a writing window in which the user can write or edit the contents of the cell. This window should include a palette of commonly entered symbols such as "*" or formula names that are hard to recognize. This allows the user to enter an unambiguous symbol by tapping an appropriate button.

A formula bar area should behave more like the word-processor application just described. If the area is empty, the application should display the writing window automatically. Naturally, the recognizer should be configured according to the type of input expected, whether text, numerical, or whatever.

Annotation, including quick notes and diagrams, represents an ideal usage of a pen in a spreadsheet application. Often, annotated text can be kept as ink and does not require recognition. Anchoring and targeting annotations on a spreadsheet is somewhat easier than on a word-processing document because cells do not flow in the same manner as text. Annotations should be anchored to the data and not the cell. This ensures that, if the data moves to another cell, the annotation moves with it.

Because spreadsheets contain much data that is not based on words, on-screen keyboards should be easy to display (and automatic in many circumstances). As with word-processing applications, writing in a spreadsheet will most likely be limited to small editing and format changes.

The user will often wish to edit the contents of a single cell to see how it affects the rest of the spreadsheet. To facilitate this operation with a pen, the application should anticipate as much as possible. For example, double-tapping a field containing a numeric value should display a numeric on-screen keypad.

If the spreadsheet offers text entry with recognition, it should provide an appropriate tool accessible from a toolbar or menu. When the user selects text-entry mode, the spreadsheet should enlarge (zoom), allowing the user to write comfortably within a cell. The application can rely on the automatic targeting capabilities provided by the Pen application programming interface (API), described in Chapter 2, to route text appropriately to different cells.

Mail

Perhaps the most common operations in electronic mail with a pen are navigation, annotation, responding to and forwarding mail, and composition of short notes.

When the user selects "new" to create a new blank message or presses a "reply" or "forward" button, the application should automatically display a writing window in which the user can enter new text. In the case of the reply and forward operations, the writing window should include the text of the original message. The user may wish to selectively edit or delete parts of the original message when replying or forwarding. The application should provide action handles for this.

Forms

Pen computers are ideal for filling out electronic forms. Forms applications can achieve superior handwriting recognition for several reasons:

- The bedit control with its built-in guides serves perfectly in many if not all the writing sections of a form.
- Because each writing window of an electronic form often expects input of a certain type, the forms application can constrain recognition to that type. For example, an application can restrict recognition to numerals for a control window that expects a telephone number.
- Through the use of word lists such as local street names or regional cities, a recognizer can greatly improve accuracy for certain input sections of the form.

Forms usually appear blank or nearly blank by default, so the most common operation in a forms application is adding text. The system's automatic targeting routes input to the proper control window without intervention by the application.

A writing window should appear automatically on the appropriate field if recognized text produces questionable results. Double-tapping should display a writing window, and the application should provide a lens (writing window) and on-screen keyboard accelerator buttons. It should also provide insertion-point action handles for selection.

Shell

The most common shell operations involve selecting files, opening files or running programs, dragging filenames to copy or move files, and deleting files. A pointer-based interface serves all such operations well. Therefore, the pen should behave as a pointer in the shell. Tapping or dragging a rectangular marquee should select files, double-tapping should open files, and dragging should copy or move files. The shell can provide deletion services on selected files either through a "delete" option in a menu or by handling a cut gesture.

A Sample Pen Application

This chapter describes a simple pen-based application called PENAPP that demonstrates some of the programming techniques covered in the previous chapters. The source code in this chapter is fragmentary, illustrating only the most interesting parts of the application. For the complete source listing, see the file PENAPP.C in the Microsoft Win32 Software Development Kit, in the SAMPLES\PEN\PENAPP directory.

PENAPP uses the sample recognizer SREC, described in the next chapter, "Writing a Recognizer." The source files for SREC also reside in the SAMPLES directory. To see PENAPP in action, you must first build both PENAPP.EXE and SREC.DLL using the supplied makefiles. Place the SREC.DLL file in your Windows directory or in a directory on the PATH list before running PENAPP.

The SAMPLES\PEN directory also provides source code for other sample pen-based applications. All code is commented, demonstrating different approaches to different issues.

Overview of PENAPP

PENAPP is a standard pen-based application with the familiar Windows look. It displays a main window with a border, an application menu, and Minimize and Maximize buttons. It also has three child windows titled Input, Info, and Raw Data.

In operation, PENAPP accepts pen input through the Input child window. Depending on the menu option, PENAPP sends the input to the system recognizer or the sample custom recognizer SREC, or collects the data into an **HPENDATA** object to create a mirror image of the ink.

The output from the recognizers is displayed through the Raw Data and Info child windows. The Raw Data child window redisplays the raw input data, sized for the smaller window. The Info child window displays one of the following, depending on the current menu selection:

- If the selected recognizer is the system recognizer, the Info child window displays the recognized ANSI text.
- If the selected recognizer is the sample custom recognizer, the Info child window displays an arrow indicating the compass direction of the input stroke. If the pen rests on the tablet surface without moving, the Info window displays a single dot.
- If the Mirror option is selected, the Info window displays a mirror image of the drawing.

In this chapter, PENAPP function and variable names appear in monospace font, including the application's entry function, `WinMain`. In keeping with the conventions of the rest of the book, API elements are styled bold.

Initialization

The PENAPP function WinMain is a standard Windows entry function. PENAPP uses initialization functions, called InitApp and InitInstance, to create windows and initialize data. To an experienced programmer in Windows, the WinMain and initialization functions will look very familiar.

WinMain

The WinMain function performs the same tasks as a regular Windows function:

- It calls the initialization functions to register window classes and create windows.
- It enters a message loop to process messages from the application queue.
- The message loop ends when the user chooses Exit from the menu, generating a WM_DESTROY message, which in turn posts a WM_QUIT message to WinMain. When the **GetMessage** function detects the WM_QUIT message, it returns NULL to end the loop.

The WinMain function calls **GetSystemMetrics** to check whether pen services are installed. If they are not found, the application should either exit with an explanatory message or alter its behavior to run without pen input.

Note that Microsoft pen services must be installed when Windows starts. Simply linking PENWIN.DLL and loading it at runtime is not sufficient to initialize pen services. For this reason, unless it is known that the application being developed will always be run on a system with pen services installed, pen API should be called through function pointers. This mechanism insures that a pen-aware application can run only on a system on which pen services has been properly installed and will not run on any system that merely has PENWIN.DLL on the path. See the HFORM sample application for an example of this technique.

For the sake of simplicity and readability, the PENAPP application described in this chapter links directly to PENWIN.DLL and does not use function pointers.

```
int PASCAL WinMain(
                        HANDLE hInstance,       // Instance handle
                        HANDLE hPrevInstance,   // Previous instance handle
                        LPSTR lpszCommandLine,  // Command line string
                        int cmdShow )                 // ShowWindow flag
{

     MSG               msg;

     // Mention to prevent compiler warnings ....
     lpszCommandLine;

     if (!hPrevInstance)                              // If first instance,
           if (!InitApp( hInstance ))          //   register window class
                 return FALSE;                       // Exit if can't
register


     if (!InitInstance( hInstance, nCmdShow ))
                                                       // Create this
instance's window
           return FALSE;                       // Exit if error


     if (!GetSystemMetrics( SM_PENWINDOWS ))         // If no pen services
           return FALSE;                                        // exit

     while (GetMessage( (LPMSG) &msg, NULL, 0, 0) )
     {
           TranslateMessage( (LPMSG) &msg );
```

```
            DispatchMessage( (LPMSG) &msg );
      }

      return 0;                                                    // Success
}
```

The InitApp function initializes data and registers the window classes. Following standard programming practice for Windows, the function returns FALSE if it cannot register the window classes.

For the Input window, InitApp specifies a cursor type of IDC_PEN. This is the default cursor type supplied by the pen-aware display driver. For more information on pen types, see the reference section for the IDC_ constants in Chapter 13, "Pen Application Programming Interface Constants."

The following fragment shows how InitApp creates the Input window:

```
BOOL InitApp( HANDLE hInstance )         // Instance handle
{
      WNDCLASS    wc;
            .
            .
            .
      //
      // Register PenApp child window classes
      //
      wc.hCursor       = LoadCursor( NULL, IDC_PEN );
      wc.hIcon         = NULL;
      wc.lpszMenuName  = NULL;
      wc.lpszClassName = (LPSTR) szPenAppInputClass;
      wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
      wc.style         = CS_VREDRAW | CS_HREDRAW | CS_SAVEBITS;
      wc.lpfnWndProc   = InputWndProc;
      if (!RegisterClass( (LPWNDCLASS) &wc) )
            return FALSE;
            .
            .
            .
}
```

InitInstance

The InitInstance function initializes all data structures for the program instance and creates the necessary windows. InitInstance looks like a standard Windows initialization function except that it calls the **InstallRecognizer** function to load the default recognizer SREC. Windows loads the default recognizer automatically, so a program should call **InstallRecognizer** to load only recognizers other than the default recognizer.

The following fragment taken from the InitInstance function shows how the program loads the SREC recognizer:

```
BOOL InitInstance(
                HANDLE hInstance,       // Instance handle
                int cmdShow )                 // ShowWindow flag
{
            .
            .
            .
     //
     // Load sample recognizer SREC
     //
     vhrec = InstallRecognizer( (LPSTR) szSampleRec );
     if (vhrec)
           return TRUE;
     else
     {
        MessageBox( vhwndMain, "Could not install recognizer SREC",
               szPenAppWnd, MB_OK | MB_ICONSTOP );
        return FALSE;
     }
}
```

Window Procedures

PENAPP provides a separate procedure to handle messages for each of its four windows. In accordance with usual programming techniques for Windows, each window procedure passes all unprocessed messages to **DefWindowProc** for default processing.

The following sections describe the procedures for the Main, Input, Raw, and Info windows.

MainWndProc

As its name implies, the MainWndProc function handles messages sent to the program's parent (main) window. These messages signal that the user has made a menu selection, resized the window, or changed the pen system through Control Panel. These actions generate, respectively, the following messages:

- WM_COMMAND message. If the user selects a command from the Options menu, the program notes the selection in the global flag viMenuSel. The state of this flag determines whether subsequent input goes to the system recognizer or the sample recognizer SREC, or is collected into an **HPENDATA** object for conversion to a mirror image.
- WM_SIZE message. PENAPP resizes its windows according to the information in the lParam variable.
- WM_PENMISCINFO message. The procedure allows this message to fall through to **DefWindowProc** for default handling.

```
LRESULT CALLBACK       MainWndProc(
                                                  HWND hwnd,              //
Window handle
                                                  UINT message,           //
Message
                                                  WPARAM wParam,          //
Varies
                                                  LPARAM lParam )         //
Varies
{
     LONG        lRet = 0L;
           .
           .
           .
     switch (message)
     {
           case WM_COMMAND:
                 switch (wParam)
                 {
                       case miExit:
                             DestroyWindow( vhwndMain );
                             break;

                       case miSample:
                       case miSystem:
                       case miMirror:
                             ResetWindow( wParam );
                             break;
                 }
                 break;
                 .
                 .
                 .
           case WM_DESTROY:
                 if (vhpendata)
                 {
                       // PKPD.DLL API: function pointer not needed:
```

```
                        DestroyPenData( vhpendata );
                        vhpendata = NULL;
                  }
                  //
                  // Unload sample recognizer. (Don't
                  // unload system default recognizer.)
                  //
                  UninstallRecognizer( vhrec );
                  vhrec = NULL;
                  PostQuitMessage( 0 );
                  break;

            default:
                  lRet = DefWindowProc( hwnd, message, wParam, lParam );
                  break;
      }
      return lRet;
}
```

InputWndProc

The InputWndProc procedure receives the WM_LBUTTONDOWN message that signals the start of an input session. It distinguishes between a pen-down event and a true mouse event, using the methods described in the "Beginning an Input Session" section in Chapter 2.

When it detects the start of a pen-input session, InputWndProc calls the **DoDefaultPenInput** function to handle the chores of initialization, data collection, and inking.

As described in the "Step 3: PE_GETPCMINFO Message" section in Chapter 2, InputWndProc immediately receives a PE_GETPCMINFO submessage. The lParam variable points to an initialized **PCMINFO** structure that the system will use for recognition. The SREC recognizer requires the structure to specify a pen-up event as a condition of termination. Accordingly, InputWndProc takes this opportunity to set the PCM_PENUP flag.

InputWndProc next receives a PE_BEGINDATA submessage, as described in "Step 6: PE_BEGINDATA Message" in Chapter 2. In response, the procedure takes one of the following courses of action:

- If currently using the sample recognizer, InputWndProc creates an **HRC** object for that recognizer and specifies the **HRC** in the **TARGET** structure pointed to by lParam. This tells **DoDefaultPenInput** to use the sample recognizer SREC instead of the system default recognizer.
- If displaying a mirror image of the ink, InputWndProc creates an **HPENDATA** object for the ink and specifies it in the **TARGET** structure pointed to by lParam. This tells **DoDefaultPenInput** to collect data into the **HPENDATA** block instead of passing it to a recognizer.
- If using the default recognizer, InputWndProc simply passes the message to **DefWindowProc**, which creates an **HRC** for the default recognizer. The code also demonstrates how an application can take advantage of the convenience afforded by **DefWindowProc**, yet override any default assumptions it makes.

  When **DefWindowProc** returns, lParam still points to the **TARGET** structure, which now reflects the default assumptions. One of these assumptions limits to one the maximum number of guesses the system recognizer should return. Since PENAPP requires a maximum of five guesses, it changes the value by calling **SetMaxResultsHRC** for the **HRC** that **DefWindowProc** creates.

The PE_ENDDATA submessage informs InputWndProc that the input session has ended. The procedure collects any symbols returned by the current recognizer into an array of symbol strings called vsyvSymbol. This portion of the code does not check to see which menu option is current. It simply collects symbols into the array if available. If the Mirror option is selected, the attempt to collect recognized symbols harmlessly fails since no **HRC** exists.

After it collects the data, InputWndProc invalidates all three child windows. This sends WM_PAINT messages to each window, clearing the Input window and causing the other two windows to display their new data.

```
#define    MAX_GUESS   5           // Maximum number of guesses
#define    MAX_CHAR          20          // Maximum number of characters per
guess

// Global Variables ************************************************
HRCRESULT  vrghresult[MAX_GUESS];         // Array of results
SYV   vsyvSymbol[MAX_GUESS][MAX_CHAR];    // Array of symbol strings
int              vcSyv[MAX_GUESS];              // Array of string lengths
.
.
.
```

```
LRESULT CALLBACK       InputWndProc(
                                                HWND hwnd,              //
Window handle
                                                UINT message,           //
Message
                                                WPARAM wParam,          //
Varies
                                                LPARAM lParam )         //
Varies
{
     LONG        lRet = 0L;        // Initialize return code to FALSE
     HRC               hrc;              // HRC object
     HDC               hdc;
     PAINSTRUCT  ps;
     DWORD       dwInfo;
     int               i, cGuess;

     switch (message)
     {
                .
                .
                .
      case WM_LBUTTONDOWN:
     //
     // Two possibilities exist: user is using mouse or the pen.
     // The latter case indicates the user is starting to write.
     //
           dwInfo = GetMessageExtraInfo();
           if (IsPenEvent( msg, dwInfo ))
           {
             if (DoDefaultPenInput( vhwndInput,
                                        (UINT)dwInfo ) == PCMR_OK)
                     lRet = TRUE;
                else
                     lRet = DefWindowProc( hwnd, msg, wParam, lParam );
           }
           break;

     case WM_PENEVENT:
           switch (wParam)
           {
             case PE_GETPCMINFO:
                //
                // If using SREC recognizer, ensure session ends
                // on pen-up.
                //
                if (viMenuSel == miSample)
                            ((LPPCMINFO) lParam)->dwPcm |= PCM_PENUP;
                lRet = DefWindowProc( hwnd, msg, wParam, lParam );
                break;

                case PE_BEGINDATA:
                //
                // 1) If using sample recognizer, create an HRC
```

```
                  //    for it and specify it in the TARGET structure
                  //    pointed to by lParam.  This tells
                  //    DoDefaultPenInput to use the sample recognizer
                  //    instead of the system default.
                  //
                  // 2) If displaying mirror image of ink, create an
                  //    HPENDATA for it.  This tells DeDefaultPenInput
                  //    to collect data into the HPENDATA object
                  //    instead of passing it to a recognizer.
                  //
                  // 3) If using default recognizer, pass to
                  //    DefWindowProc.  DefWindowProc sets the maximum
                  //    number of guesses to 1; the code below shows
                  //    how to access the HRC that DefWindowProc
                  //    creates and reset the maximum number of
                  //    guesses to MAX_GUESS.
                  //
                        if (vhpendata)
                        {
                              DestroyPenData( vhpendata );
                              vhpendata = NULL;
                        }

                        switch (viMenuSel)
                        {
                              case miSample:
                                    hrc = CreateCompatibleHRC( NULL, vhrec );
                                    if (hrc)
                                    {
                                          ((LPTARGET) lParam)->dwData = hrc;
                                          lRet = LRET_HRC;
                                    }
                                    break;

                              case miMirror:
                                    vhpendata = CreatePenData( NULL, 0,

PDTS_HIENGLISH, 0 );
                                    if (vhpendata)
                                    {
                                          ((LPTARGET) lParam)->dwData =
vhpendata;
                                          lRet = LRET_HPENDATA;
                                    }
                                    break;

                              case miSystem:
                                    lRet = DefWindowProc( hwnd, msg,

wParam, lParam );
                                    //
                                    // On return, lParam->dwData points to
HRC.
                                    // Use it to reset max number of guesses.
                                    //
```

```
                                    SetMaxResultsHRC(
                                                          ((LPTARGET) lParam)-
>dwData,
                                                                MAX_GUESS );
                                    break;
                              }
                              break;

                        case PE_ENDDATA:
                        //
                        // DefWindowProc will destroy vhpendata, so if
                        // collecting mirror image, don't let DefWindowProc
                        // handle message.
                        //
                           if (viMenuSel != miMirror)
                           lRet = DefWindowProc(hwnd, msg, wParam, lParam);
                           break;

                        case PE_RESULT:
                        //
                        // At end of input, collect recognition results (if
                        // any) into symbol strings.  DoDefaultPenInput
                        // generates the PE_RESULT submessage only when
                        // using a recognizer.  The lParam contains the HRC
                        // for the recognition process.
                        //
                        // Collect pen data for DrawRawData
                              vhpendata = CreatePenDataHRC( (HRC) lParam );

                        // Initialize array to zero
                              for (i = 0; i < MAX_GUESS; i++)
                                          vcSyv[i] = SYV_NULL;

                        // Get number of guesses available
                              cGuess = GetResultsHRC( (HRC) lParam,
                                                            GRH_ALL,
                                                            (LPHRCRESULT)
vrghresult,
                                                            MAX_GUESS );

                        // Get guesses (in vsyvSymbol) and
                        // their lengths (in vcSyv)
                              if (cGuess != HRCR_ERROR )
                                    for (i = 0; i < cGuess; i++)
                                          vcSyv[i] = GetSymbolsHRCRESULT(

vrghresult[i],
                                                                  0,
                                                                  (LPSYV)
vsyvSymbol[i],
                                                                  MAX_CHAR );
                              break;
                              .
                              .
                              .
```

```
            default:
                  lRet = DefWindowProc( hwnd, msg, wParam, lParam );
            }                                                       // End
switch (wParam)
     break;

     default:
            lRet = DefWindowProc( hwnd, message, wParam, lParam );

  }                                                                 // End
switch (msg)
     return lRet;
}
```

InfoWndProc

PENAPP displays results in the Info window. When the InfoWndProc procedure receives a WM_PAINT message, it calls one of three functions to display the results, depending on the current menu selection. It calls:

- DisplayGuesses if the Default option is chosen.
- DrawArrow if the Sample option is chosen.
- DrawMirrorImage if the Mirror option is chosen.

The following list describes the three functions and points out interesting portions of their code.

DisplayGuesses

The DisplayGuesses function writes the guesses returned from the default recognizer. The guesses appear in a column, listed in descending order of confidence. The **for** loop shown below converts the symbol values to characters, then calls the Windows function **TextOut** to display the text.

```
VOID DisplayGuesses( HDC hdc )                        // DC handle
{
     TEXTMETRIC  tm;
     int               nX, nY;                                          //
Text coords
           .
           .
           .
     for (i = 0; i < MAX_GUESS; i++)
     {
           if (vcSyv[i])
           {
                 SymbolToCharacter( (LPSYV) vsyvSymbol[i],
                                          vcSyv[i],
                                          (LPSTR) szText,
                                          (LPINT) &cChar );
                 TextOut( hdc, nX, nY, (LPSTR) szText, cChar );
                 nY += tm.tmExternalLeading + tm.tmHeight;
           }
     }
}
```

DrawArrow

DrawArrow draws an arrow to indicate the symbol value returned from the SREC recognizer. SREC returns a compass direction determined from the start and end points of the stroke. DrawArrow reads the direction in vsyvSymbol and diplays an appropriate arrow.

DrawMirrorImage

DrawMirrorImage creates a mirror image of the data by subtracting each x-coordinate from the tablet width. This moves each point from one side of the tablet to a corresponding position on the other side. In other words, the original distance from the tablet's left side now becomes the point's distance from the tablet's right side.

DrawMirrorImage works in five steps:

1. Creates a duplicate **HPENDATA** of the input data.
2. Calls **TrimPenData** to remove unneeded information from the block.
3. Converts the new **HPENDATA** block to a mirror image.
4. Displays the mirror image by calling **DrawPenDataFmt**.
5. Deletes the mirror image **HPENDATA** block.

RawWndProc

The RawWndProc function is a standard Windows procedure for the Raw Data child window. It calls the DrawRawData function to draw a copy of the input resized for the Raw Data window. Normal pen-down strokes appear in the current window color; pen-up strokes appear in blue. Note that the system recognizer GRECO.DLL does not collect pen-up strokes. Therefore, the blue pen-up strokes do not appear when default recognition is selected from the menu.

DrawRawData calls the **DrawPenDataEx** function to display the strokes. Since **DrawPenDataEx** does not show pen-up strokes, DrawRawData first changes all pen-up strokes to pen-down strokes. The following fragment illustrates this:

```
VOID DrawRawData( HDC hdc )
{
     PENDATAHEADER   pendataheader;  // Header for vhpendata
     HPEN           hpenUp, hpenSave;    // GDI pen for up-strokes
     UINT            fPen;           // Pen flag
     UINT            iStroke=0;      // Stroke counter
     int             nWidth;         // Ink width
           .
           .
           .
     if ( !GetPenDataInfo( vhpendata, &pendataheader, NULL, 0 ))
           return;

     nWidth   = NSetExtents (hdc, &pendataheader, &rectWnd );
     hpenUp   = CreatePen( PS_SOLID, nWidth, rgbBlue );
     hpenSave = SelectObject( hdc, hpenUp );

     // Loop for each stroke, beginning with first
     for (iStroke = 0; iStroke < pendataheader.cStrokes; ++iStroke)
     {
     //
     // If down stroke, use same ink characteristics as original.
     // If up stroke, first call SetStrokeAttributes to convert it
     // to a down stroke, then draw it in blue ink with GDI pen.
     //
      if (GetStrokeAttributes(vhpendata, iStroke, NULL, GSA_DOWN))
                 fPen = DPD_DRAWSEL;
       else
       {
           SetStrokeAttributes( vhpendata, iStroke, 1, SSA_DOWN );
           fPen = DPD_HDCPEN;
       }

       iRet = DrawPenDataEx(hdc, NULL, vhpendata, iStroke, iStroke,
                                           0, IX_END, NULL, NULL, fPen );

      // Set altered strokes back to their original pen-up state
      if (fPen == DPD_HDCPEN)
        SetStrokeAttributes( vhpendata, iStroke, 0, SSA_DOWN );
     }

     SelectObject( hdc, hpenSave );
```

```
        DeleteObject( hpenUp );
        return
}
```

Writing a Recognizer

A recognizer is a dynamic-link library (DLL) that interprets lines of ink as characters and symbols. Version 2.0 of the Pen API allows a pen-based application to install multiple recognizers and use them selectively. Each recognizer should specialize in recognizing a particular set of symbols instead of trying to handle many different types. Besides keeping the recognizer code manageable, this approach lets an application choose among several available recognizers to fulfill its current recognition needs.

The recognizer developer must know both sides of the interface between application and recognizer. The foregoing chapters, particularly Chapter 5, "The Recognition Process," should be read before venturing into this one.

Such a developer should also have some familiarity with the coding requirements of a DLL. For information about how to write a DLL, see the *Guide to Programming* manual in the Microsoft Windows Software Development Kit. In addition, the "Writing a Dynamic-Link Library for Windows" chapter in the MASM version 6.1 *Programmer's Guide* offers valuable information about DLL coding requirements.

This chapter describes the framework of a recognizer DLL and the functions it must export. The final section presents a sample recognizer called SREC. The source file for SREC resides in the SAMPLES\PEN\SREC subdirectory.

Recognizer Objects

Three objects serve the process of recognition, identified by their handles: recognition context (**HRC**), recognition context result (**HRCRESULT**), and word list (**HWL**). The structure and implementation of these objects are left to the recognizer developer and remain invisible to applications and the system. However, the objects must comply with the following two requirements:

- The handle value must be a 32-bit pointer to the object in memory.
- The first DWORD (32 bits) of the memory that the handle points to is reserved for system use. The recognizer must not alter the DWORD value during the life of the object.

Thus, a recognizer's internal structure of a recognition object should be of the following form:

```
typedef struct {
     DWORD dwReserved;
           .
           .
           .
     } INTERNALOBJECT;
```

It is the application's responsibility to destroy the recognition objects when finished. A recognizer should validate all handles to ensure an object exists before processing. Although a product of an **HRC**, an **HRCRESULT** is usually an independent object. Destroying an **HRC** does not destroy its **HRCRESULT** objects, which remain valid objects and must be destroyed separately.

A single **HWL** object can be associated with multiple **HRC** objects at any given time. The recognizer should not allow alteration of an **HWL** while processing any of its associated **HRC** objects. Similarly, the recognizer should not allow destruction of an **HWL** before the destruction of all its associated **HRC** objects. In either case, the recognizer should return an error to the application.

How a Recognizer Works

There are two techniques for recognizing handwriting, called *bitmap* and *vector recognition*.

Bitmap recognition attempts to match an ink image with a record of known character images. The bitmap recognizer sees the ink data as a stencil pattern of points that it can compare to a library of patterns, searching for the closest match. This technique, employed by optical character recognizers (OCRs), works well for patterns limited to a few styles and sizes.

In contrast, vector recognition sees the ink as lines rather than points. The method considers characteristics of the lines collected as the pen moves. These characteristics include sequence, curvature, direction, and so forth. Given the wide varieties and styles of handwriting, vector recognition works best for deciphering pen input. The Pen API does not mandate which method a recognizer employs, but is designed to facilitate vector rather than bitmap recognition.

List of Exported Functions

Technically, an application does not call directly into a recognizer's exported functions, though the distinction is not important to the recognizer. Instead, all calls go to the system, which acts as a switchboard to route the calls to the proper recognizer when more than one recognizer is installed. For example, when an application loads a recognizer with **InstallRecognizer**, that recognizer exports functions with the same name as those exported by other recognizers, including the default recognizer. The system automatically transfers calls to the correct recognizer based on the **HRC** argument or other value that identifies the intended recipient.

As a DLL, a recognizer must export functions to the pen system that installs the recognizer. This section lists these recognition functions, describes their purpose, and identifies them as required or optional.

The optional functions that a recognizer should provide depend on the clients it will serve. Commonly, a recognizer DLL is part of an application package designed only for that application. As author of both client application and its private DLL, the developer need write only those functions the application requires. In this case, the developer is also free to design other functions not specified by the Pen API.

At the other extreme, some recognizers serve client applications indiscriminately. For example, a developer might market a recognizer of foreign script to various application developers with international product lines. A recognizer can also take on the role of system default recognizer, in which case it must:

- Export all recognizer functions.
- Recognize letters, punctuation, numbers, and predefined gestures from ink data.
- Associate raw data with matched results.
- Return characters only within the requested subset.
- Return an "I don't know" response when appropriate.

A system default recognizer should support all recognizer functions, not merely export stub versions of the optional functions. At the very least, the recognizer must return an HRCR_UNSUPPORTED value from functions it does not support. For a description of how to specify the system recognizer through the registry, see Appendix A, "Differences Between Versions 1.0 and 2.0 of the Pen Application Programming Interface."

The following sections describe all 47 recognition functions of the Pen API version 2.0. The lists of functions represent the following categories:

- Initialization
- **HRC** functions
- **HRCRESULT** functions
- Alphabet support
- Word lists
- Training

All recognizers must export 13 required functions, which are indicated by asterisks in the following tables. The functions without asterisks are optional for a recognizer. Each table corresponds to one of the categories listed previously, with functions arranged in alphabetic order within the table.

Initialization

The Pen API specifies the following functions for initializing, modifying, and closing down the recognizer. Note that, in version 2.0 of the Pen API, the required function **ConfigRecognizer** handles all initialization and configuration tasks. The other initialization functions are obsolete in version 2.0 and should only be included in a recognizer if it is expected to work with older applications that work with a version 1.0 recognizer (see the Microsoft Pen Windows, version 1.0 documentation for descriptions of these functions).

Function	Description
ConfigRecognizer*	Provides access for querying or altering recognizer configuration. In version 2.0, only the system calls **ConfigRecognizer**. Applications call **ConfigHREC**, which the system translates into a call to **ConfigRecognizer**.
CloseRecognizer	Required only for compatibility with version 1.0 API. Called when the system uninstalls a version 1.0 recognizer.
InitRecognizer	Required only for compatibility with version 1.0 API. Called when the system installs a version 1.0 recognizer.
RecognizeDataInternal	Required only for compatibility with version 1.0 API. The system calls this function only when an application calls **RecognizeData**.
RecognizeInternal	Required only for compatibility with version 1.0 API. The system calls this function only when an application calls **Recognize**.

HRC Functions

In general terms, the **HRC** functions carry out the recognition process. Together, they collect raw data, derive recognized symbols from the data, and place the symbols into **HRCRESULT** objects. Their work ends with recognition. To retrieve the recognized symbols, an application calls the **HRCRESULT** functions described in the next section.

Function	Description
AddPenInputHRC*	Adds input to the recognizer's **HRC** object. This function is normally called at every pen movement, providing data a few points at a time.
CreatePenDataHRC	Returns an **HPENDATA** handle for the pen data within the **HRC**.
CreateCompatibleHRC*	Creates an empty **HRC** object, ready to receive input data.
DestroyHRC*	Frees the memory occupied by an **HRC** object, invalidating the handle value.
EnableGestureSetHRC	Enables or disables the recognition of a specified set of gestures.
EnableSystemDictionaryHRC	Specifies whether or not the recognizer should use its dictionary to validate recognition guesses.
EndPenInputHRC*	Notifies the recognizer that input has ended for the session. This function does not initiate recognition of the collected data; the client application must call ProcessHRC to do that.
GetBoxResultsHRC	Gets recognition results for a range of boxes.
GetGuideHRC	Retrieves a copy of the GUIDE structure (if any) used in the **HRC**.
GetHRECFromHRC*	Gets the module handle to the recognizer DLL attached to the **HRC**.
GetMaxResultsHRC	Gets the current maximum number of guesses the recognizer can make for the **HRC**.
GetResultsHRC*	Retrieves recognition results as **HRCRESULT** objects. Each object represents one guess.
ProcessHRC*	Tells the recognizer that it should begin recognition and sets the maximum amount of time allowed for the task.
SetGuideHRC	Specifies a GUIDE structure for recognition.
SetMaxResultsHRC	Sets the maximum number of guesses the recognizer can make for the **HRC**.

HRCRESULT Functions

The **HRCRESULT** functions retrieve recognized symbols and other associated information from the recognizer. The **GetResultsHRC** function described in the previous table collects results into one or more **HRCRESULT** objects. Each object represents an alternative interpretation the recognizer has made about the input. Once an application calls **GetResultsHRC** to create the **HRCRESULT** objects, it can then call the **HRCRESULT** functions listed here to get the recognized characters from the objects.

Function	Description
CreateInksetHRCRESULT	Creates an inkset corresponding to recognition results.
DestroyHRCRESULT*	Frees the memory occupied by an **HRCRESULT** object, invalidating the handle value.
GetAlternateWordsHRCRESULT	Retrieves alternative guesses from the results of a recognition process.
GetBoxMappingHRCRESULT	Retrieves the indices for a range of symbols in boxes. For example, if writing begins in the fifth box of a guide, this function returns the index 4 for the first symbol.
GetHotspotsHRCRESULT	Returns the critical point for a given recognized gesture. (See the "Hot Spots" section.)
GetSymbolCountHRCRESULT*	Gets the length of the symbol array that forms one of the recognizer's guesses.
GetSymbolsHRCRESULT*	Retrieves symbol values corresponding to one of the recognizer's guesses.

Specifying an Alphabet Set

By supporting the following alphabet functions, a recognizer enables an application to specify which alphabet sets to consider during recognition. Alphabets are sets of characters within the entire range of characters the recognizer can interpret. For example, an application can limit recognition to any combination of lowercase characters, punctuation, math symbols, and so forth.

The Pen API defines ALC_ values to identify an alphabet set. For further information on alphabets and an abbreviated list of the most common ALC_ values, see "Configuring the HRC" in Chapter 5, "The Recognition Process." A full list of ALC_ values appears in Chapter 13, "Pen Application Programming Interface Constants."

The Pen API allows an application to set a priority when using multiple alphabet sets. Priority can resolve conflicts when one glyph has different interpretations in different alphabets. For example, consider a case in which input consists of both letters and numerals, but the application expects numerals more often. By setting an alphabet of ALC_ALPHANUMERIC and a priority of ALC_NUMERIC, the application tells the recognizer to consider both letters and numerals, but interpret for numerals first. This helps resolve the problem of distinguishing between the numeral "0" and the letter "O."

The following table lists the optional recognizer functions that pertain to alphabets.

Function	Description
GetAlphabetHRC	Retrieves bitwise-OR flags of ALC_ values indicating which alphabet(s) the recognizer can currently recognize.
GetAlphabetPriorityHRC	Retrieves bitwise-OR flags of ALC_values indicating priority.
SetAlphabetHRC	Constrains recognition to a specified set of alphabet characters.
SetAlphabetPriorityHRC	Specifies the priority of alphabets used during recognition.
SetBoxAlphabetHRC	Constrains recognition to a set of specified alphabet characters for individual boxes in a group of boxes.

Word Lists

A word list acts as a broad alphabet set. A list consists of valid words that can influence the confidence a recognizer places in a guess. After guessing at a word (or phrase), a recognizer can search for the guess in one or more word lists. Locating a guess in a word list helps verify the validity of the guess.

A word list can consist of a small group of words permanently stored in the recognizer's data segment. Often, however, the words reside in an accompanying file that the recognizer reads as required. A word list file should be in standard ANSI text format, one word per line, with each line ending in a carriage return and linefeed. This allows the user to emend the files, if necessary, with a text editor.

A recognizer that uses word lists should export the **ReadHWL** and **WriteHWL** functions. These functions read and write standard word list files, enabling an application to move words directly between a file and an **HWL** object.

The table below lists the exported functions for a recognizer that uses word lists.

Function	Description
AddWordsHWL	Adds a specified collection of words to an existing word list in memory.
CreateHWL	Creates a word list in memory, either empty or containing a given list of words.
DestroyHWL	Destroys a word list, invalidating the handle.
GetWordlistCoercionHRC	Retrieves the current degree of influence a word list or the system dictionary has on the confidence level of a guess.
GetWordlistHRC	Retrieves a word list from the **HRC** object.
ReadHWL	Reads from a file into an empty word list. The words must be in ANSI text format, one word per line, each line ending with a carriage return and linefeed.
SetWordlistCoercionHRC	Specifies the influence a word list or the system dictionary should exert on the confidence level of a guess.
SetWordlistHRC	Sets a word list into the **HRC** object.
WriteHWL	Writes from a word list to a file. The words are written as ANSI text, one word per line.

Training

Training is optional for a recognizer and the method of its implementation is up to the developer. Through training, a recognizer can consider the individual style and writing characteristics of different users when interpreting handwriting.

Training can be classified as either passive or active. However, the distinction usually pertains more to the application than to the recognizer. In passive training, the application quietly calls the recognizer's training functions whenever the user corrects a wrong guess. Correctly implemented, passive training helps ensure that the recognizer learns from its mistakes.

Active training takes place only when specifically requested by the user. A training window prompts the user for written samples, then the verified input is given to the recognizer to store in its database for that user. The recognizer can provide the active training support, though usually this task is left to an application. Microsoft usability studies have shown that users do not object to the time invested in active training.

The following table lists the functions that a recognizer with training capabilities can export. Only **TrainHREC** is used by version 2.0 Pen API. The other functions are obsolete in version 2.0 and should be included in a recognizer only if it is expected to work with older applications that work with a version 1.0 recognizer (see the Microsoft Pen Windows version 1.0 documentation for descriptions of these functions).

Function	Description
TrainContextInternal	Passes a previous recognition result that may contain errors along with the required interpretation. The system calls this function in response to a call to **TrainContext**. This function applies only to training recognizers compatible with version 1.0 of the Pen API. It is superseded by the **TrainHREC** function.
TrainHREC	Passes ink data and its required interpretation to the recognizer. The recognizer then stores the data and interpretation for future reference.
TrainInkInternal	Passes a previous erroneous recognition result along with the correct interpretation. The system calls this function when the application calls **TrainInk**. **TrainInkInternal** is rendered obsolete by **TrainHREC** and is only for training recognizers compatible with version 1.0 of the Pen API.

Interpreting Input

Typically, a recognizer converts pen input to recognized data in three steps:

1. Collect and process the raw pen input data.
2. Segment the written symbols.
3. Note the order and direction of pen strokes.

Processing Raw Data

Raw data for recognition consists of pen coordinates. At a minimum, the recognizer must collect coordinate data while the pen is in contact with the tablet. Optionally, the recognizer can also process additional pen data such as pen pressure, the height of the pen tip above the pad, the angle of the pen, and the rotation of the pen. Not all pen devices can provide such information.

The Microsoft Handwriting Recognizer (GRECO.DLL) processes only coordinate data. The Pen API provides the **OEMPENINFO** structure for other types of pen data. For details, see the entry for **OEMPENINFO** in Chapter 11, "Pen Application Programming Interface Structures."

Noise Reduction and Normalization

To improve recognition, a recognizer can optionally employ *noise reduction* and *normalization* techniques. Noise reduction filters the input to weed out erroneous input—for example, pen skips, inadvertent marks from the user, or spurious noise from the input device.

Normalization corrects the natural skewing of handwritten text. In the same way that lines of text tend to run askew on blank paper, lines of pen input are usually not parallel to the top and bottom of the tablet. (An application can provide guidelines to help correct this tendency.)

Coordinates should be normalized relative to a horizontal line, called the *baseline*, that marks the bottom of the text. The baseline is analogous to a single line on lined notebook paper. Letter descenders, such as the lower parts of "y" or "j," descend below the baseline.

If a guide is present, its vertical coordinate defines the baseline. The baseline of a single-line guide in absolute coordinates is the sum of the **yOrigin** and **cyBase** members of the **GUIDE** structure. For more information, see the reference description of the GUIDE structure.

Allowed Time

The recognizer must return within the period of time specified by the *dwTimeMax* parameter of the **ProcessHRC** function. This parameter can have the values PH_MIN, PH_DEFAULT, or PH_MAX. Respectively, these values limit the time allowed for processing to approximately 50 milliseconds, 200 milliseconds, or as much time as required.

For values other than PH_MAX, the recognizer must ensure that it does not exceed the allotted time. The recognizer can either regularly poll with the **GetTickCount** function to mark the passage of time or, through the **SetTimer** function, provide a callback function that sets a time-out flag. The SREC sample recognizer described at the end of this chapter demonstrates the latter technique.

Allowed Number of Guesses

The recognizer must return no more than the maximum number of guesses specified by the **SetMaxResultsHRC** function. For a description of this function, see the "Number of Recognition Guesses" section in Chapter 5. By default, the recognizer returns only its best guess with no alternative guesses.

Segmentation of Symbols

A recognizer can view symbols at any granularity. For instance, most handwriting recognizers see individual letters and numerals as symbols. A recognizer for cursive writing, on the other hand, may see a complete word as a single symbol without distinguishing each letter of the word.

No matter how it views symbols, a recognizer must separate them within a stream of written symbols, a process called *segmentation*. The task of segmenting letters is greatly facilitated if the application provides box guides. In this case, the recognizer can assume that strokes lying within a box constitute a single character. The problem of accurate segmentation becomes more difficult for unguided text.

Segmentation is a crucial issue for recognizing different handwriting styles. The following table lists the forms of input in decreasing order of constraint on the user. The information in the table is taken from IBM Research Report RC 11175, No. 50249, (May 21, 1985), *An Adaptive System for Handwriting Recognition*, by C. C. Tappert.

Input form	Definition
Boxed input	Each character appears within its own box.
Discrete spaced	A set of strokes in a given space belong to the same character. (This is also called *external segmentation*.)
Discrete run on	Printed characters can overlap.
Cursive	Letters are connected by ligatures. The recognizer must either identify discrete letters or interpret a whole word at a time.
Mixed	The recognizer can segment discrete, run-on, and cursive writing.

Figure 8.1 illustrates these various styles.

{ewc msdncd, EWGraphic, bsd23553 0 /a "SDKIMB.BMP"}

The Pen API places few restrictions on the recognizer. At a minimum, however, a default recognizer must be able to recognize discrete characters because many applications do not use boxed input.

Stroke Order and Direction

Noting the order and placement of strokes can help a recognizer handle the following cases:

- Delayed strokes. A delayed stroke occurs after other strokes, but belongs to an earlier, unfinished character. For example, in writing the word "two," the user might cross the "t" only after writing the rest of the word.
- Correction strokes. A correction stroke alters the interpretation of other strokes – for example, placing a small stroke on the top of a "y" to change it to a "g." Correction strokes are often delayed.
- Characters written out of order. For example, the user should be able to first write "t o," then fill in a "w" between the letters. The recognizer should recognize the completed word as "two" instead of "tow."
- Variations in stroke order or direction. Different users often write the multiple strokes of characters in a different order and direction. To take an extreme example, the four strokes forming a capital "E" can be written in $2^{(4)} * 4! = 384$ distinct ways.

Returning Results

To return results, the recognizer must conform to the procedures described in the section "Getting Results" in Chapter 5. That section examines the case where the words "clear," "dear," "clean," "dean," and "deer" represent valid interpretations of a handwritten word. In such a case, the recognizer should return the possibilities arranged in order of decreasing likelihood.

Without an internal concept of likelihood, the recognizer must impose an arbitrary order. However, for multiple recognizers to cooperate, a recognizer must have some concept of a poor match and be able to return "unknown" in lieu of a guess. While the pen API does not strictly require a recognizer to assign confidence levels to its guesses, without confidence values a recognizer cannot work efficiently with other recognizers.

The recognizer must be able to associate individual strokes with a recognized symbol. Applications can use the stroke data to correctly juxtapose the recognized text with the ink on the screen, redraw the data, or send information to other recognizers.

Speed and timing are very important in the recognition process. A recognizer should recognize input at least at the speed of normal handwriting, approximately two to three characters per second.

Results Messages

Results messages concerning recognition come from the system, not the recognizer. The messages depend on what services the application uses:

- If **DoDefaultPenInput** runs the recognition process, it sends a chain of messages to the application as described in Chapter 2, "Starting Out with System Defaults."
- If the application calls one of the version 1.0 recognizer functions, such as **Recognize** or **ProcessWriting**, the system generates a WM_RCRESULT message. The system can send many WM_RCRESULT messages during a single recognition event, depending on the frequency that the application has specified for receiving data. The *lParam* of each message points to a new, self-contained **RCRESULT** structure that contains the recognition results generated since the last WM_RCRESULT message.

The RCRESULT Structure

A recognizer can store its results in any format the developer wishes. It need not create an **RCRESULT** structure except in response to calls to certain superseded functions such as **Recognize** and **RecognizeData**. For completeness, this section describes the **RCRESULT** structure, which the developer may wish to use as a model for storage. Although a recognizer must calculate the information found in an **RCRESULT** structure, it need not organize the information in the same format.

Note The **RCRESULT** structure is not required in version 2.0 of the Pen API and is supported only to maintain compatibility with older applications that use version 1.0 recognizer API.

The first member of the **RCRESULT** structure is a list called the *symbol graph*, which contains all the recognizer's guesses. An application can read the guesses in order of likelihood by walking through the symbol graph.

The Symbol Graph

The best way to understand the symbol graph is to first diagram its contents before describing how to actually read it. The following discussion again takes up the example in the section "Getting Results" in Chapter 5, in which the recognizer generated the five guesses "clear," "dear," "clean," "dean," and "deer." A diagram of the symbol graph that represents all these possibilities might look like this:

```
Letter:      {  cl   |   d  }    e    {  a   |   e  }    {  r   |   n  }
Confidence:     80%      60%    100%     85%     20%        80%     50%

Average:     clear   86%
             dear    81%
             clean   79%
             dean    74%
             deer    58%
```

If you study the diagram a moment, you will see its logic. Alternative letters appear separated by a C bitwise-OR symbol, with the most likely alternative first. All the guesses in this example, however, agree that the second (or third) character is the letter "e," so it has no alternatives. Taking the first letter in each alternative set produces the most likely of the guesses – in this case, the word "clear."

The symbol graph includes confidence values for each character or character set (as in the case of the interpretation "cl"). The recognizer can determine a confidence value for an entire word by averaging the values for each character or character set, as shown in the diagram above. (Note that this hypothesis is purely for purposes of discussion. The pen API does not mandate how a recognizer determines its confidence levels. The influence of word lists and other factors may also change confidence levels.)

Symbol graphs must, therefore, contain three types of information:

- All characters (or character sets) determined as likely interpretations for a set of pen strokes
- A map identifying the pen strokes that correspond to each interpretation
- A confidence level for each interpretation

As described in Chapter 11, "Pen Application Programming Interface Structures," the symbol graph is a structure of type **SYG**. The **SYG** structure contains two additional data structures that provide the needed information: symbol correspondence and symbol element structures.

A symbol correspondence structure **SYC** delineates a specific subset of the strokes entered by the user. Each **SYC** contains the first and last strokes of a subset; these strokes and the strokes between them

define the subset of pen data associated with the **SYC**. The symbol graph contains an array of **SYC** structures, each of which corresponds to a different part of the ink input. Taken together, the **SYC** structures define all the ink gathered during the input session.

A **SYG** structure also contains an array of symbol element **SYE** structures. An **SYE** contains a symbol value, a confidence level, and an index into the array of **SYC** structures. Each character or character set in the recognized input has its own symbol element.

The Best Guess

The **RCRESULT** structure also provides information about the recognizer's "best guess." The best guess is simply the first interpretation in the symbol graph, which lists interpretations in descending order of probability. Since an application is often interested only in the most likely interpretation, the recognizer should place in the **RCRESULT** the following three members specifically to identify the best guess:

- The **lpsyv** member points to a null-terminated symbol string containing the best guess.
- The **cSyv** member contains the number of symbols in the best guess string.
- The **hSyv** member is the handle to the memory block to which **lpsyv** points.

Location and Position of the Input

The **RCRESULT** structure also contains information regarding the location and position of the ink entered by the user.

- The **nBaseLine** member is the recognizer's estimate of the baseline of the ink entered by the user. (For a definition of baseline, see "Noise Reduction and Normalization" earlier in this chapter.) If the baseline is not known, the recognizer sets this value to 0. The Microsoft Handwriting Recognizer (GRECO.DLL) sets **nBaseLine** to 0.
- The **nMidLine** member is the recognizer's estimate of the midline of the ink entered by the user. If the midline is not known, the recognizer sets this value to 0. The Microsoft Handwriting Recognizer sets **nMidLine** to 0.
- The **rectBoundInk** member is a Windows **RECT** structure. It holds the bounding rectangle that circumscribes the area of the screen on which the user has written. Typically, an application uses **rectBoundInk** to invalidate the screen area to update the display in the appropriate location. This occurs, for example, when Windows replaces ink on the screen with recognized text.

Contextual Information

Two elements of the **RCRESULT** structure provide information about the recognition event, but not as a part of the results of recognition. They are **lprc**, a far pointer to the **RC** structure passed to the **Recognize** function, and **wResultsType**, a flag that describes how the recognition event proceeded. The **wResultsType** flag contains a combination of RCRT_ constants, described in Chapter 13, "Pen Application Programming Interface Constants."

The Ink

The final two members of the **RCRESULT** structure contain information about the ink entered by the user.

- The **pntEnd** member contains the last point of the ink data from the user only if PCM_RECTBOUND or PCM_RECTEXCLUDE have been specified. An application sets these flags either in the **lPcm** member of the **RC** structure or the **dwPcm** member of the **PCMINFO** structure.
- The **hpendata** member is a handle to a pen data memory block that contains all of the ink information entered by the user.

Hot Spots

While recognizing a symbol, the recognizer may also identify critical points on the symbol called *hot spots*. Hot spots can apply to any symbol but usually are of interest only for gestures. For example, if the user writes an X for deletion, the cross of the X – its hot spot – points to the item to be deleted. If the recognizer identifies hot spots for a recognized symbol, it places coordinates for all hot spot points in the **rgpntHotSpot** member of the symbol's **SYG** structure. This array can hold up to MAXHOTSPOT points. Note that all symbols do not have the same number of hot spots, and many symbols have none.

Writing a Recognizer

A recognizer must reflect the developer's requirements, design, and programming style. Although it cannot prescribe any one method for creating a recognizer, this section offers guidance on writing several important recognition functions. The section also presents a sample recognizer called SREC to illustrate some of the information in this chapter.

For a description of how an application uses the **InstallRecognizer** function to load a recognizer function, see "Creating the HRC" in Chapter 5. The reference section for **InstallRecognizer** in Chapter 10 also discusses how to load a recognizer by using the **LoadLibrary** function to aid in debugging the recognizer. To designate a recognizer as the system default recognizer, see "Registry Configuration" in Appendix A.

Recognition Functions

The following sections provide fragmentary examples of the recognition functions **AddPenInputHRC**, **CreateCompatibleHRC**, **CreateInksetHRCRESULT**, **CreatePenDataHRC**, and **DestroyHRC**.

For purposes of discussion, the examples assume the recognizer's **HRC** takes the form of a private structure called HRCinternal. This structure contains a handle to an **HPENDATA** block, which stores the stroke information and pen data points for an input session. (For a description of **HPENDATA**, see "The HPENDATA Object" in Chapter 4.) The **HRC** applies only to a single session – say, a word written by the user. The client application must create a new **HRC** to recognize the next word.

The following **typedef** statement defines the hypothetical **HRC** object used by the example code fragments in this section:

```
typedef struct
{
     DWORD       reserved;          // Reserve first DWORD for system
     HGLOBAL     hglobal;           // Handle from GlobalAlloc
     HPENDATA    hpendata;          // HPENDATA handle for input
     int         wPdk;              // Current stroke is pen-up or down
                                   // Other information for the HRC
           .
           .
} HRCinternal, FAR *LPHRCinternal;
```

Notice that the structure reserves the first DWORD for system use. This is required for all recognizer objects. The structure also groups pertinent variables within the object itself instead of allocating them as global data. This ensures that the object remains private to the client that creates it.

In the code that follows, all internal functions have an "N" prefix. This helps distinguish them from standard API functions.

CreateCompatibleHRC

The **CreateCompatibleHRC** function allocates memory for the **HRC** object. The example below calls the Windows function **GlobalAlloc** and uses the returned memory handle as the **HRC**. Locking the allocated memory with **GlobalLock** provides a far pointer to the allocated HRCinternal structure.

When an input session begins, the pen state is always down. Therefore, the function initializes the wPdk member to PDK_DOWN.

```
HRC WINAPI CreateCompatibleHRC( HRC hrcTemplate, HREC hrec )
{
     HGLOBAL         hglobal;        // Handle of allocated HRC object
     LPHRCinternal   lphrc;          // Far pointer to object
           .
           .
           .
     // Allocate memory for HRC and get far pointer to it
     hglobal = GlobalAlloc( GHND, sizeof( HRCinternal ) );
     lphrc   = (LPHRCinternal) GlobalLock( hglobal );

     // If failure, return NULL
     if (!lphrc)
           return NULL;

     // Save HRC memory handle, because DestroyHRC will need it
     lphrc->hglobal = hglobal;

     // If template provided, copy its information to the new HRC
     if (hrcTemplate)
           NCopyTemplateInfo( hrcTemplate, lphrc );

     // Initialize other information
     lphrc->hpendata = CreatePenData( NULL, 0, PDTS_HIENGLISH, 0 );
     lphrc->wPdk     = PDK_DOWN;
           .
           .
           .
     // If no errors, return pointer as HRC handle
     return ((HRC) lphrc);
}
```

DestroyHRC

Generally, the life of an **HRC** object is brief. An application destroys the **HRC** after obtaining results from the recognizer and creates a new **HRC** at the start of the next input session.

In the model described above, **CreateCompatibleHRC** calls **GlobalAlloc** to allocate system memory in which the **HRC** object resides. The function returns a pointer to the fixed allocation as an **HRC** handle.

DestroyHRC reverses the action, using the original **HGLOBAL** handle to free the allocated memory. When **DestroyHRC** successfully returns, the **HRC** handle is no longer valid. The application should set the handle to NULL to prevent accidental reuse, as described in "Destroying the HRC" in Chapter 5.

```
int WINAPI DestroyHRC( HRC hrc )
{
    LPHRCinternal  lphrc = (LPHRCinternal) hrc;  // Pointer to HRC

     if (!GlobalUnlock( lphrc->hglobal ))
           return HRCR_OK;
     else
           return HRCR_ERROR;
}
```

AddPenInputHRC

As the pen moves, an application continually gets data from the pen driver by calling **GetPenInput**. It then passes the retrieved information to the recognizer via **AddPenInputHRC**. This function may be called many times before the user completes the stroke. Each time, **AddPenInputHRC** receives a small subset of points that collectively form a pen stroke.

Along with the subset of points, **AddPenInputHRC** receives a **STROKEINFO** structure that represents the points in the subset. For example, the member **dwTick** contains the starting time for each subset, not the entire stroke. (The starting time of the first subset of a stroke is also the starting time of the entire stroke.) Similarly, **cPnt** contains the point count only for the current subset.

The following example lets the system take care of accumulating the points. It calls the **AddPointsPenData** API function to add the subset of points to the **HPENDATA** block belonging to the **HRC**. The code also demonstrates how a recognizer can determine when one stroke ends and another begins. This allows the recognizer to take some intermediate steps to facilitate recognition at the end of each stroke. Such intermediate work can remove some of the burden from **ProcessHRC**, improving response time and perhaps accuracy as well.

```
int WINAPI AddPenInputHRC( HRC hrc, LPPOINT lppnt, LPVOID lpvOem,
                           UINT oemdatatype, LPSTROKEINFO lpsi )
{
    LPHRCinternal   lphrc = (LPHRCinternal) hrc; // Pointer to HRC
    int             iRet  = HRCR_OK;             // Return code
            .
            .
            .
     //
     // If state change from down to up (or vice versa), the previous
     // stroke has ended and the point data that lppnt points to belongs
     // to a new stroke.  Take any intermediate action to process the
     // just-completed stroke.
     //
     if (lpsi->wPdk != lphrc->wPdk)
     {
                 .
                 .                                          // Take
intermediate action
                 .
           lphrc->wPdk = lpsi->wPdk                  // Note new pen state
     }

     // Accumulate stroke points in internal HPENDATA object
     if (!AddPointsPenData( lphrc->hpendata,   // HPENDATA handle
                                    lppnt,              // Point subset
                                    lpvOem,             // OEM data
                                    lpsi ))             // Subset
STROKEINFO
           iRet = HRCR_ERROR;
                 .
                 .
                 .
     // Return appropriate error code (HRCR_OK or HRCR_ERROR)
     return (iRet);
}
```

CreatePenDataHRC

The **CreatePenDataHRC** function returns a handle to an **HPENDATA** object that contains the raw data used for recognition. In the example code above, **AddPenInputHRC** has already done the work of storing the pen data into an internal **HPENDATA**. Thus, the hypothetical **CreatePenDataHRC** function outlined below simply duplicates the internal object.

```
HPENDATA WINAPI    CreatePenDataHRC( HRC hrc )
{
    LPHRCinternal  lphrc = (LPHRCinternal) hrc;  // Pointer to HRC

     // Clone the internal HPENDATA and return its handle
     return (DuplicatePenData( lphrc->hpendata, 0 ));
}
```

CreateInksetHRCRESULT

The optional **CreateInksetHRCRESULT** function creates a corresponding inkset from the data in an **HPENDATA** object. For a description of inksets and the **INTERVAL** structure, see "The HINKSET Object" in Chapter 4.

A description of the **CreateInksetHRCRESULT** function first requires a brief discussion of stroke start and end times. Some of this information also appears in the section "Timing Information" in Chapter 4, but it is presented here from the point of view of the recognizer rather than the application.

A stroke's start time is the starting tick count of the first group of points that **AddPenInputHRC** receives when a new stroke begins. The member **dwTick** of the **STROKEINFO** structure is the number of milliseconds that have elapsed since the system tick reference determined at system startup time. A recognizer can retrieve this value in an **ABSTIME** structure through a call to the Pen API function **GetPenMiscInfo**:

```
ABSTIME  atTickRef;
GetPenMiscInfo( PMI_TICKREF, (LPARAM)((LPABSTIME) &atTickRef) );
```

The **STROKEINFO** structure also contains in its **cPnt** member the number of points in the collection. Because the pen device sends points at a constant rate (called the *sampling rate*), the number of points in the collection implies how much time has elapsed between the first and last points.

The sampling rate does not change, so the recognizer need only determine the rate during initialization and store the value. The following example shows how a recognizer can get the sampling rate from the pen driver:

```
PENINFO     pinfo;
HDRVR       hPenDrv;
int         vnSamplingRate;
      .
      .
      .
hPenDrv = OpenDriver( "pen", 0, 0 );
if (hPenDrv)
{
      if (SendDriverMessage( hPenDrv, DRV_GetPenInfo, (LPARAM)&pinfo, 0 ))
            vnSamplingRate = pinfo.nSamplingRate;
      CloseDriver( hDriverPen, 0, 0 );
}
```

With this information, **CreateInksetHRCRESULT** can fill an **INTERVAL** structure with a stroke's start and stop times as shown below. The code assumes the **HRCRESULT** object contains the **HRC** handle. This allows the internal function NGetStrokeFromHPENDATA to locate the internal **HPENDATA** object with the raw input data.

```
HINKSET WINAPI CreateInksetHRCRESULT( HRCRESULT hrcresult,
                                                        UINT isyv, UINT csyv)
{
      HINKSET     hinkset;
      INTERVAL    interval;
      STROKEINFO  si;
      DWORD       dwMsec;
      UINT        i, j;

      // Call Pen API to create hinkset object
      hinkset = CreateInkset( GMEM_MOVEABLE | GMEM_DDESHARE );
```

```
    // Initialize INTERVAL with tick reference (described above)
    GetPenMiscInfo( PMI_TICKREF,
                   (LPARAM)((LPABSTIME) &interval.atBegin) );

    // For each SYV in the HRCRESULT between the given indices ...
    for (i = 0; i < csyv; i++, isyv++ )
    {
          j = 0;

          // For each stroke in the SYV ...
          while ( NGetStrokeFromHPENDATA( hrcresult, &si, isyv, j++ ))
          {
                //
                // Calculate the interval for the stroke.
                // Note si.dwTick is the number of milliseconds
                // that have elapsed since system start-up.
                //
                dwMsec = (DWORD)(1000L*interval.atBegin.sec +
                                 interval.atBegin.ms + si.dwTick);
                interval.atBegin.sec = dwMsec/1000L;
                interval.atBegin.ms  = (UINT)(dwMsec % 1000L);

                dwMsec = (DWORD)(1000L*interval.atBegin.sec +
                                  interval.atBegin.ms +
                                  1000L*si.cPnt/vnSamplingRate);
                interval.atEnd.sec = dwMsec/1000L;
                interval.atEnd.ms  = (UINT)(dwMsec % 1000L);

                // Call Pen API function to add interval to inkset
                AddInksetInterval( hinkset, (LPINTERVAL) &interval );
          }
    }
    return (hinkset);
}
```

A Sample Recognizer

This section describes a simple recognizer called SREC that demonstrates some of the information given in this chapter. The text describes the most interesting parts of the program and illustrates with code fragments. The complete source listing for SREC.C resides in the SAMPLES\PEN\SREC subdirectory.

SREC is used by the PENAPP application described in Chapter 7, "A Sample Pen Application." To see how SREC works, you must create both PENAPP.EXE and SREC.DLL using the supplied MAKE files, then run PENAPP.

When using the SREC recognizer, PENAPP specifies that a stroke ends when the pen leaves the tablet. Therefore, SREC recognizes only one stroke at a time. SREC takes the beginning and ending points of the stroke and calculates the nearest compass direction of the line formed by these endpoints.

For its **HRC** object, SREC creates a structure that contains an **HPENDATA** handle to the input data, the module handle returned from **InstallRecognizer**, and recognition results. The following **typedef** statements define the **HRC** and **HRCRESULT** objects for SREC. Notice that SREC keeps its **HRCRESULT** within the **HRC.**

```
typedef struct                          // HRCRESULT object
{
     DWORD   reserved;                   // Reserve top DWORD
     SYG     syg;                        // Recognition results
} HRCRESULTinternal, FAR *LPHRCRESULTinternal;

typedef struct                          // HRC object
{
     DWORD               reserved;    // Reserve top DWORD
     HPENDATA            hpendata;    // Raw pen data to be recognized
     HREC                hrec;        // Module handle for SREC
     HRCRESULTinternal   hrcresult;   // HRCRESULT structure
} HRCinternal, FAR *LPHRCinternal;
```

When it finishes recognizing a stroke, SREC fills out a **SYG** symbol graph structure. The structure holds one of the symbol values listed here:

Symbol value	Direction
syvEast	Right
syvSouth	Down
syvWest	Left
syvNorth	Up
syvDot	Single tap

The following sections describe the functions that SREC exports. These functions appear under the same categories described earlier in this chapter, in the section "List of Exported Functions." This allows for quick cross-referencing between a general description of a function and its actual implementation in SREC.

Although defined by the Pen API, the function names below appear in monospace font rather than bold because the names refer to routines in the SREC.C source file.

SREC Initialization Functions

As a Windows dynamic-link library, SREC exports **LibMain** and **WEP**. As a recognizer, it also exports the required initialization function **ConfigRecognizer**. All recognizers compatible with version 2.0 of the Pen API must provide these functions.

LibMain and WEP

The first two functions in the SREC recognizer are the standard Windows functions required in any dynamic-link library, **LibMain** and **WEP**. **LibMain**, the main DLL function, is analogous to **WinMain**. It performs any needed initialization and unlocks the data segment of the library. **WEP** is the standard DLL termination function, which receives control when Windows unloads the DLL. For a description of **WEP**, see the references listed at the beginning of this chapter.

ConfigRecognizer

The **ConfigRecognizer** function handles the recognizer's initialization tasks and configures it for special options. When it loads a recognizer, **InstallRecognizer** internally calls the recognizer's **ConfigRecognizer** function with the subcommand WCR_INITRECOGNIZER. In response to this call, the recognizer should perform any required initialization tasks.

As its name suggests, **ConfigRecognizer** handles more than initialization work. It also provides the means for setting recognizer options and to query for capabilities. With version 2.0 of the Pen API, which can load multiple recognizers, applications do not call **ConfigRecognizer**, because the function provides no way to identify the intended library. Instead, applications call the **ConfigHREC** function, which takes

the same arguments as **ConfigRecognizer**, with the addition of the **HREC** handle returned from **InstallRecognizer**. Internally, the system identifies the intended recognizer from the handle and passes the arguments to **ConfigRecognizer** in the appropriate recognizer. Thus, **ConfigHREC** and **ConfigRecognizer** refer to the same function. **ConfigRecognizer** is unique in that it is the only function exported by a recognizer that applications do not call directly.

As the following code fragment shows, SREC returns only its identification string and version number from **ConfigRecognizer**. Note also that SREC does not allow itself to be set as the system recognizer. Since SREC does not support standard editing gestures or recognize characters, it cannot serve as a system default recognizer.

```
int WINAPI   ConfigRecognizer( UINT uSubFunc,
                               WPARAM wParam, LPARAM lParam )
{
     int         iRet = TRUE;

     switch ( uSubFunc )
     {
                 .
                 .
                 .

           case WCR_INITRECOGNIZER:            // No initialization or
           case WCR_CLOSERECOGNIZER:     //   clean up duties to
                 break;                              //   perform

           case WCR_RECOGNAME:
                 lstrncpy( (LPSTR)lParam, szID, wParam );
                 break;

           case WCR_DEFAULT:             // Can't be system default
```

```
            case WCR_QUERY:                     // Does not support config
dialog
            case WCR_QUERYLANGUAGE:       // Does not support any language
                  iRet = FALSE;
                  break;

            case WCR_PWVERSION:
            case WCR_VERSION:
                  iRet = 0x0002;                // Recognizer version 2.0
                  break;

            default:
                  iRet = FALSE;                 // Anything else is
unsupported
                  break;
      }
      return iRet;
}
```

For a complete list of WCR_ subfunctions, refer to the reference section for **ConfigRecognizer** in Chapter 10.

When the last client application unloads a recognizer, the **UninstallRecognizer** function calls the recognizer's **ConfigRecognizer** function with the command WCR_CLOSERECOGNIZER. This informs the recognizer that it is being unloaded. The previous code takes no action for WCR_CLOSERECOGNIZER because in SREC, memory allocations come from the local heap. As with any Windows-based program, a DLL's heap resides in its data segment. When Windows unloads a DLL, it automatically returns the entire data segment to the memory pool.

However, unloading SREC does not destroy its internal **HPENDATA** object. **HPENDATA** blocks occupy global heap space. If the client application terminates or unloads SREC without first destroying all **HRC** objects created by SREC, the corresponding **HPENDATA** blocks are left orphaned in memory. A recognizer more intelligent than SREC should maintain a count of active **HPENDATA** allocations and free any that remain before terminating.

A recognizer's **WEP** routine also receives control when Windows unloads the recognizer. Developers should note a subtle difference between handling cleanup chores in **ConfigRecognizer** and in **WEP**. When the former executes in response to the WCR_CLOSERECOGNIZER subfunction, the client is still active. However, the **WEP** routine cannot safely make the same assumption when it executes. **ConfigRecognizer** can therefore conceivably post a message to the client or perform some other action that relies on an active recipient.

The disadvantage of **ConfigRecognizer** is that the recognizer cannot be certain the function will execute because the client might not call **UninstallRecognizer**. Since the **WEP** function is guaranteed to execute when Windows unloads the recognizer, essential cleanup duties, such as unhooking interrupts, should be handled in **WEP**.

SREC Recognition Functions

This section takes a brief look at some of SREC's exported recognition functions, including **CreateCompatibleHRC**, **ProcessHRC**, and **CreatePenDataHRC**. The code uses the macro

```
#define  lpHRC    ((LPHRCinternal) hrc)
```

to represent a far pointer to the **HRC** object.

CreateCompatibleHRC

The **CreateCompatibleHRC** function allocates an HRCinternal structure in the local heap, creates an **HPENDATA** block for the pen data, and returns a far pointer to the structure. The LPTR argument forces **LocalAlloc** to return a far pointer to the allocation instead of a memory handle. This far pointer serves as SREC's **HRC** handle.

Since the **HRC** has no configurable elements, SREC ignores any template **HRC** provided in the first parameter.

```
HRC WINAPI CreateCompatibleHRC( HRC hrcTemplate, HREC hrec )
{
     HRC         hrc;

     hrc = (HRC) LocalAlloc( LPTR, sizeof( HRCinternal ) );
     if (hrc)
     {
           lpHRC->hrec     = hrec;
           lpHRC->hpendata = CreatePenData( NULL, 0, PDTS_HIENGLISH, 0 );
           if (lpHRC->hpendata)
                 return (hrc);
     }
     LocalFree( (HLOCAL) hrc );          // If error, free allocation
     return NULL;                                    //   and return NULL
}
```

ProcessHRC

The most interesting feature of SREC's **ProcessHRC** function is the way it sets a time limit for processing. If called with a limit of PH_MIN or PH_DEFAULT, **ProcessHRC** passes the address of a callback function to **SetTimer**. When the specified time-out period elapses, the callback function receives control and sets a global flag called vfOutOfTime.

A recognizer can use this technique to ensure that it does not overrun a specified time limit. Its internal processing functions should check the vfOutOfTime flag regularly and, if it is set, terminate immediately. In this case, **ProcessHRC** returns a value of HRCR_INCOMPLETE to tell the caller recognition has not yet finished.

```
int WINAPI      ProcessHRC( HRC hrc, DWORD dwTimeMax )
{
    UINT        idTimer, uTime;
    int         iRet;

     vfOutOfTime = FALSE;                           // Initialize time-out
flag
     if (dwTimeMax != PH_MAX)                       // If time limit
specified ...
     {
           uTime   = (dwTimeMax == PH_MIN) ? 50 : 200;
```

```
            idTimer = SetTimer( NULL, NULL, uTime, (TIMERPROC) TimerProc );
            iRet    = GetSYG( hrc );                    // Quit if out of time
            KillTimer( NULL, idTimer );
      }
      else
            iRet = GetSYG( hrc );               // Don't quit until finished

      return (iRet);
}


VOID CALLBACK    TimerProc( HWND hwnd, UINT ms, UINT iId, DWORD dwTm )
{
      vfOutOfTime = TRUE;
}
```

CreatePenDataHRC

SREC keeps an **HPENDATA** handle in its **HRC** structure. Because the **AddPenInputHRC** function has already stored pen input in the internal **HPENDATA** block, **CreatePenDataHRC** simply duplicates the block.

```
HPENDATA WINAPI  CreatePenDataHRC( HRC hrc )
{
      if (hrc)
            return (DuplicatePenData( lpHRC->hpendata, 0 ));
      else
            return NULL;
}
```

Summary of the Pen Application Programming Interface

This chapter summarizes the pen services by listing them according to category. The lists complement the detailed descriptions of functions, structures, messages, and constants in the reference chapters that follow this chapter. The lists let you quickly identify those services that pertain to your application, then refer to the reference chapters for detailed information.

The "Pen Kernel Functions" section is of interest to developers who want to write applications for Microsoft Windows 95 that use ink data without the presence of pen hardware.

Pen API Functions

The Pen Application Programming Interface (API) provides functions that can be divided into 10 broad categories. The following table describes the 10 categories. Other tables list the functions within each category.

Function category	Description
System and hardware	Provide information about pen hardware and current system assumptions.
Display	Display ink data, get screen information.
Pen data	Collect, copy, move, and delete data in an **HPENDATA** object.
Recognition	Recognize handwritten characters.
Symbol manipulation	Collect and convert symbols returned from a recognizer.
Time intervals	Manipulate time intervals associated with pen strokes.
Compression	Reduce the size of an **HPENDATA** object.
Utility	Miscellaneous utility services provided by the system.
Hook	Program hooks to monitor inking or recognition.
Obsolete	Obsolete functions of version 1.0 maintained by version 2.0 only for compatibility reasons.

List of Pen API Functions

The following tables list by category all functions in version 2.0 of the Pen API. Functions appear in alphabetical order within each category, together with a brief description.

System and hardware functions	Description
GetPenAsyncState	Gets state of pen barrel button.
GetPenMiscInfo	Gets current system settings.
GetVersionPenWin	Gets the Pen API version number.
SetPenMiscInfo	Sets system defaults and assumptions.
UpdatePenInfo	Called by the pen driver to notify the system of a change in the driver configuration.

Display functions	Description
CreatePenDataRegion	Returns a screen region that contains the points of an **HPENDATA** object.
DrawPenDataEx	Enhanced version of DrawPenData.
DrawPenDataFmt	Default version of DrawPenDataEx.
RedisplayPenData	Displays collected pen data exactly as originally drawn.
ShowKeyboard	Displays or hides the on-screen keyboard.
StartInking	Begins the process of leaving a visible ink trail as the pen moves. See the descriptions of StartPenInput and DoDefaultPenInput.
StopInking	Stops the inking process begun by a call to StartInking.

Pen data functions	Description
AddPenDataHRC	Adds an **HPENDATA** object to an **HRC**.
AddPointsPenData	Adds new points and original equipment manufacturer (OEM) data to an existing **HPENDATA** object.
CreatePenData	Allocates memory for a new **HPENDATA** object and initializes its header.
CreatePenDataEx	Enhanced version of CreatePenData.
CreatePenDataHRC	Returns handle to **HPENDATA** object associated with an **HRC**.
DestroyPenData	Frees memory occupied by an **HPENDATA** memory block.
DuplicatePenData	Clones an existing **HPENDATA** object.
ExtractPenDataPoints	Copies or removes points from a stroke.
ExtractPenDataStrokes	Copies or removes selected strokes, optionally creating a new **HPENDATA** object from the copied strokes.

GetPenDataAttributes	Retrieves information about an **HPENDATA** object.
GetPenDataInfo	Gets status information for an **HPENDATA** object.
GetPointsFromPenData	Returns an array of points from an **HPENDATA** object.
GetStrokeAttributes	Retrieves information about a stroke.
GetStrokeTableAttributes	Retrieves information about a stroke's class. The class is an entry in a table stored in the PENDATAHEADER structure.
InsertPenData	Combines two **HPENDATA** blocks.
InsertPenDataPoints	Inserts points into a stroke in an **HPENDATA** object.
InsertPenDataStroke	Inserts data for a new stroke into an existing **HPENDATA** object.
OffsetPenData	Offsets pen data points by a specified amount.
PenDataFromBuffer	Reverse of PenDataToBuffer, which must be called first. Creates an **HPENDATA** block and writes the buffer back into it.
PenDataToBuffer	Serializes the contents of an **HPENDATA** block to a buffer.
RemovePenDataStrokes	Removes specified strokes from an **HPENDATA** object.
ResizePenData	Scales ink data to fit a specified rectangle.
SetStrokeAttributes	Sets attributes of a stroke. Reverse of GetStrokeAttributes.
SetStrokeTableAttributes	Sets attributes for a stroke's class. Reverse of GetStrokeTableAttributes.

Recognition functions	Description
AddPenInputHRC	Adds raw pen input to an **HRC** object.
AddWordsHWL	Adds words to a word list.
ConfigRecognizer	System access to recognizer configuration. Applications should not call this function.
CreateCompatibleHRC	Creates an empty **HRC** object.
CreateHWL	Creates a word list.
DestroyHRC	Destroys a recognizer's recognition context object.
DestroyHRCRESULT	Destroys a recognizer's results object.
DestroyHWL	Destroys the word list handle created by CreateHWL and frees its memory.
EnableGestureSetHRC	Enables or disables recognition of specified gestures.
EnableSystemDictionaryHRC	Specifies whether a recognizer should use its dictionary.

EndPenInputHRC	Informs a recognizer that the input session has ended.
GetAlphabetHRC	Retrieves the current alphabet from a recognizer.
GetAlphabetPriorityHRC	Retrieves the current alphabet priority from a recognizer.
GetAlternateWordsHRCRESULT	Gets alternative guesses made by a recognizer.
GetBoxMappingHRCRESULT	Retrieves from a recognizer the locations of a range of symbols in boxes.
GetBoxResultsHRC	Gets recognition results for a range of boxes.
GetGuideHRC	Gets a copy of the GUIDE structure (if any) in an **HRC** object.
GetHotspotsHRCRESULT	Returns the hot spots for a specified gesture.
GetHRECFromHRC	Gets module handle of recognizer from an **HRC**.
GetMaxResultsHRC	Gets the maximum number of guesses a recognizer can make.
GetResultsHRC	Retrieves an **HRCRESULT** object from recognizer containing recognition results.
GetWordlistCoercionHRC	Gets the current degree of influence a word list or dictionary has on recognition confidence levels.
GetWordlistHRC	Gets a word list from an **HRC** object.
InstallRecognizer	Loads a specified recognizer.
ProcessHRC	Tells recognizer to process input for a given period of time.
ReadHWL	Reads a word list from a file.
SetAlphabetHRC	Specifies the alphabet for a recognition session.
SetAlphabetPriorityHRC	Specifies alphabet priority for a session.
SetBoxAlphabetHRC	Specifies the alphabet for a range of boxes.
SetGuideHRC	Specifies guides for an **HRC**.
SetMaxResultsHRC	Sets the maximum number of guesses a recognizer can make.
SetWordlistCoercionHRC	Sets the degree of influence a word list or dictionary has on recognition confidence levels.
SetWordlistHRC	Identifies a word list for an **HRC** object.
TrainHREC	Passes ink and correct interpretations to recognizer for training.
UninstallRecognizer	Unloads a specified recognizer.
WriteHWL	Writes a word list to a file.

Symbol manipulation	Description

functions	
CharacterToSymbol	Converts an ANSI string to an array of symbol values.
EnumSymbols	Enumerates symbol strings in a symbol graph.
FirstSymbolFromGraph	Returns the array of symbols that is the most likely interpretation of a specific symbol graph.
GetSymbolCount	Returns the number of symbol strings contained in the symbol graph.
GetSymbolCountHRCRESULT	Gets the number of symbol values in results.
GetSymbolMaxLength	Gets the length of the longest symbol string contained in the symbol graph.
GetSymbolsHRCRESULT	Gets symbol values of recognition results.
SymbolToCharacter	Converts an array of symbols to an ANSI string.

Time interval functions	Description
AddInksetInterval	Adds an INTERVAL structure to an existing **HINKSET** object.
CreateInkset	Creates an empty inkset into which intervals can be added with the AddInksetInterval function.
CreateInksetHRCRESULT	Retrieves the intervals for a specified series of symbols returned by the recognizer.
DestroyInkset	Frees memory occupied by an inkset and invalidates the **HINKSET** handle.
GetInksetInterval	Copies a series of intervals from an **HINKSET** object to an array of INTERVAL structures.
GetInksetIntervalCount	Returns the number of intervals in an **HINKSET** object.

Compression functions	Description
CompressPenData	Compresses and uncompresses data.
DPtoTP	Converts display coordinates to tablet coordinates.
MetricScalePenData	Converts pen data points to one of the supported metric modes.
TPtoDP	Converts tablet coordinates to display coordinates.
TrimPenData	Removes selected data from an **HPENDATA** block.

Utility functions	Description
AtomicVirtualEvent	Blocks out physical pen events while posting virtual events.
BoundingRectFromPoints	Returns the bounding rectangle of an

	array of points.
ConfigHREC	Configures or queries recognizer options.
CorrectWriting	Displays lens or Correct Text dialog box to allow user to correct text.
CorrectWritingEx	Sends text to the CorrectText dialog box to allow the user to edit text using the Japanese Data Input Window. (Japanese version only.)
DoDefaultPenInput	Runs high-level recognition/data collection. Internally calls **StartPenInput**, **StartInking**, **StopPenInput**, and **StopInking**.
GetPenAppFlags	Returns the task flags created by **SetPenAppFlags**.
GetPenInput	Collects input data as the user writes.
GetPenResource	Retrieves a copy of the pen services resource. (Japanese version only.)
HitTestPenData	Determines whether a given point lies near a stroke.
IsPenEvent	Determines whether a WM_LBUTTONDOWN message is generated by a mouse or pen device.
KKConvert	(Japanese version only.) Activates the Kana-to-Kanji converter.
PeekPenInput	Retrieves information about a pen packet in the pen input queue. This function is similar to **GetPenInput**, but does not remove the pen packet from the queue.
PostVirtualKeyEvent	Simulates a keystroke by sending a virtual key code to Windows.
PostVirtualMouseEvent	Simulates mouse activity by sending a virtual mouse event to Windows.
SetPenAppFlags	Sets pen flags for the application that are used globally by the pen services.
StartPenInput	Begins collecting into an internal buffer ink data generated by the moving pen. See also the descriptions of . **DoDefaultPenInput** and **StartInking**.
StopPenInput	Ends collection process begun by a call to **StartPenInput**.
TargetPoints	Determines the logical recipient of data among several targets.

Hook functions	**Description**
SetPenHook	Sets or removes a hook for capturing low-level pen events.
SetResultsHookHREC	Sets a hook for recognition results.

UnhookResultsHookHREC	Unhooks a hook set by **SetResultsHookHREC**.

Obsolete functions	Description
BeginEnumStrokes	Locks an **HPENDATA** memory block in global memory in preparation for reading.
CloseRecognizer	Called by the system when uninstalling a recognizer. Subfunction has been superseded by WCR_CLOSERECOGNIZER in **ConfigRecognizer**.
CompactPenData	Data compression function superseded by **CompressPenData** and **TrimPenData**.
DictionarySearch	Searches dictionary for a word or phrase.
DrawPenData	Displays ink according to a display context **HDC**. Superseded by **DrawPenDataEx**.
EmulatePen	Emulates a pen system.
EndEnumStrokes	Unlocks an **HPENDATA** memory block. Required after calling **BeginEnumStrokes**.
GetGlobalRC	Retrieves a copy of the current system **RC** structure.
GetPenDataStroke	Gets the raw data for a stroke stored in an **HPENDATA** memory block.
GetPenHwEventData	Retrieves a range of pen event data from the internal pen data buffer.
InitRC	Initializes recognition context for the recognizer. Only for compatibility with version 1.0.
InitRecognizer	Called by the system when it installs a recognizer. Superseded by WCR_INITRECOGNIZER subfunction in **ConfigRecognizer**.
IsPenAware	Checks application's capability to handle pen events. Superseded by **GetPenAppFlags**.
ProcessWriting	Runs high-level recognition services. Superseded by **DoDefaultPenInput**.
Recognize	Begins recognition for a version 1.0 recognizer.
RecognizeData	Delayed recognition for a version 1.0 recognizer.
RegisterPenApp	Identifies an application to the system as pen-aware. Superseded by **SetPenAppFlags**.
SetGlobalRC	Sets default settings for the specified recognition context. This function

	should be called only from the pen Control Panel program.
SetRecogHook	Installs and removes a recognition hook in version 1.0. Superseded by **SetResultsHookHREC**.
TrainContext	Passes to the recognizer a previous recognition result that may contain errors along with the required interpretation.
TrainContextInternal	Called by system when an application calls **TrainContext**.
TrainInk	Informs the recognizer at the DLL recognition level that the raw data input represents the symbol value results.
TrainInkInternal	Called by system when an application calls **TrainInk**.

Pen Kernel Functions

As described in Chapter 1, the services of the Pen API are provided by the libraries PENWIN.DLL and PKPD.DLL. PENWIN.DLL is provided by original equipment manufacturers and exists only on systems with attached pen hardware. The ink management services of PKPD.DLL, however, are part of Windows 95. This allows an application to display and manipulate ink data with any installation of Windows 95, even one without pen hardware.

The following table lists the 41 Pen API functions exported by PKPD. If an application detects Windows 95 without PENWIN.DLL, it can still use these functions to display, examine, alter, and compress existing ink data.

AddInksetInterval

AddPointsPenData

BeginEnumStrokes

BoundingRectFromPoints

CompactPenData

CompressPenData

CreateInkset

CreatePenData

CreatePenDataEx

CreatePenDataRegion

DestroyInkset

DestroyPenData

DrawPenData

DrawPenDataEx

DrawPenDataFmt

DuplicatePenData

EndEnumStrokes

ExtractPenDataPoints

ExtractPenDataStrokes

GetInksetInterval

GetInksetIntervalCount

GetPenDataAttributes

GetPenDataInfo

GetPenDataStroke

GetPointsFromPenData

GetStrokeAttributes

GetStrokeTableAttributes

HitTestPenData

InsertPenData

InsertPenDataPoints

InsertPenDataStroke

MetricScalePenData

OffsetPenData

PenDataFromBuffer

PenDataToBuffer

RedisplayPenData

RemovePenDataStrokes

ResizePenData

SetStrokeAttributes

SetStrokeTableAttributes

TrimPenData

Pen API Structures

The Pen API defines 31 structures in the following categories:

Structure category	Description
System and hardware	Information about the system and pen hardware.
Display	Structures that affect display.
Guides and controls	Structures that affect boxes, guides, and controls.
Recognition	Pertain to the process and results of recognition.
Pen data	Information about points and strokes.
Target	Pertain to target windows.
Time intervals	Stroke interval information.

The following tables list the structures of the Pen API by category. For structures new to version 2.0, the first member is **cbSize**, which contains the structure's size in bytes.

Important Before using a version 2.0 structure, an application must initialize its **cbSize** mem-ber with the value **sizeof(** *structname* **)**, where *structname* represents the name of the structure. For example:

```
INKINGINFO        inkinginfo;
inkinginfo.cbSize = sizeof( INKINGINFO );
```

or

```
INKINGINFO        inkinginfo = {sizeof( INKINGINFO )};
```

System and hardware structures	Description
CALBSTRUCT	Pen calibration information.
OEMPENINFO	Tablet hardware information provided by original equipment manufacturer.
PDEVENT	Provides information about the pen device associated with an IN_PDEVENT notification.
PENINFO	Pen or tablet hardware information.

Display structures	Description
ANIMATEINFO	Animation information used by the DrawPenDataEx function.
CWX	Specifies optional parameters for the CorrectWritingEx function. (Japanese version only.)
INKINGINFO	Specifies where and how to display ink.
PCMINFO	Specifies screen areas that affect pen data collection.
PENTIP	Width and color of ink trail left by pen.
RECTOFS	Offsets of inflated or deflated writing

	area.
SKBINFO	Information about on-screen keyboard.

Guide and control structures	Description
BOXEDITINFO	Size information for boxed edit control.
BOXLAYOUT	Layout of boxed edit control.
CTLINITBEDIT	Initialization for boxed edit control.
CTLINITHEDIT	Initialization for handwriting edit control.
CTLINITIEDIT	Initialization for ink edit control.
GUIDE	Characteristics of handwriting guides.

Recognition structures	Description
BOXRESULTS	Results returned from the **GetBoxResultsHRC** function.
RC	Various information about the recognition context used by version 1.0 recognition functions.
RCRESULT	Results of recognition initiated through a version 1.0 recognition function.
SYC	Symbol correspondence linking ink strokes with a particular recognized symbol.
SYE	Symbol element containing a recognized symbol and its confidence level.
SYG	Symbol graph containing **SYC** and **SYE** struc-tures that together specify all guesses a recog-nizer has made.

Pen data structures	Description
PENDATAHEADER	Header structure of an **HPENDATA** memory block.
PENPACKET	Data sent by pen driver to inform system of pen activity.
STRKFMT	Attributes of a stroke.
STROKEINFO	Information about points making up a single stroke.

Time interval structures	Description
ABSTIME	Time of a pen data point in seconds and milliseconds.
INTERVAL	Start and end times for a set of data points.

Target structures	Description
INPPARAMS	Describes a set of targets.
TARGET	Information about a single target window.
TARGINFO	Information about a set of targets.

Pen API Messages

The Pen API defines message and submessage values identified by the following prefixes:

Message prefix	Description
CTLINIT_	Submessages for WM_CTLINIT.
DRV_	Messages from the pen hardware driver.
HE_	Submessages of WM_PENCTL for hedit and bedit controls.
HN_	Notification messages for hedit and bedit controls.
IE_	Messages for iedit control.
IN_	Notification messages for iedit control.
PE_	Submessages for WM_PENEVENT.
PMSC_	Submessages for WM_PENMISC.
SKB_	Submessages for WM_SKB.
SKN_	Notifications for WM_SKB.
WM_	Window messages for pen-based applications.

Pen API Constants

The PENWIN.H header file defines manifest constants for the Pen API, most of which begin with prefixes of two or more letters to indicate their purpose. The following table describes the prefixes of the Pen API constants:

Constant prefix	Description
AI_	Options for **AnimateProc** function.
ALC_	Alphabet codes.
BEI_	Information for bedit control.
BESC_	Size of bedit control.
BXD_	Define dimensions of bedit control (Roman).
BXDK_	Define dimensions of bedit control (Japanese).
BXS_	Styles for bedit controls.
CMPD_	Options for **CompressPenData** function.
COLOR_	Input method editor colors for bedit control.
CPD_	Storage codes for **CreatePenDataEx**.
CPDR_	Types for **CreatePenDataRegion**.
CWR_	Options for **CorrectWriting**.
DIRQ_	Dictionary request codes.
DPD_	Flags for **DrawPenDataEx** function.
EPDP_	Options for **ExtractPenDataPoints**.
EPDS_	Options for **ExtractPenDataStrokes**.
GGRC_	Options for **GetGlobalRC**.
GPA_	Options for **GetPenDataAttributes**.
GRH_	Return types from **GetResultsHRC**.
GSA_	Options for **GetStrokeAttributes**.
GST_	Codes for **EnableGestureSetHRC**.
HEKK_	Subfunctions for kana-kanji conversions.
HEP_	Subfunctions for HE_STOPINKMODE.
HKP_	Options for **SetPenHook**.
HRCR_	Return values from recognition functions.
IDC_	Cursor types defined by pen display driver.
IEB_	Codes for background in iedit controls.
IEDO_	Codes for draw option IE_ messages.
IEM_	Menu codes for IE_ iedit control messages.
IEMODE_	Codes for IE_SETMODE message.
IEN_	Codes for IE_SETNOTIFY message.
IER_	Codes for stroke format IE_ messages.
IEREC_	Codes for recognition IE_ messages.
IES_	Style attributes for iedit control.
IESEC_	Codes for security IE_ messages.
IESF_	Flags for **STRKFMT** structure.
ISR_	Return values from inkset functions.
OBM_	Public bitmaps.

PCM_	Termination conditions for pen collection mode.
PCMR_	Return values from data collection functions.
PDC_	Pen device capability codes.
PDK_	Pen driver state bits for **GetPenAsyncState** function.
PDR_	General pen data return values.
PDT_	Pen driver values specific to original equipment manufacturer.
PDTS_	Trim options for **MetricScalePenData**.
PDTT_	Trim options for **CompactPenData**.
PENTIP_	Values for **PENTIP** structure.
PHW_	Report codes for **CreatePenDataEx**.
PII_	Flags for **INKINGINFO** structure.
PMI_	Codes for **GetPenMiscInfo**, **SetPenMiscInfo**, and WM_PENMISCINFO.
PMSCL_	*lParam* values for PMSC_ constants.
PMSCR_	Return values for PMSC_TARGETING subfunction.
PWF_	Subcodes of PMI_SYSFLAGS.
RC_	Values for **RC** structure.
RCD_	Indicates writing direction (left to right, top down, etc.).
RCO_	Recognition options for **RC** structure.
RCOR_	Tablet orientation codes.
RCP_	User preference codes.
RCRT_	Values for **wResultsType** member of **RCRESULT** structure.
REC_	Return codes from a version 1.0 recognizer.
RHH_	Hook types for **ResultsHookHREC**.
SGRC_	Options for **SetGlobalRC**.
SHC_	Codes for word-list coercion functions.
SKB_	Flag values for **ShowKeyboard**.
SSA_	Options for **SetStrokeAttributes**.
SSH_	Indicates writing direction (left to right, etc.).
SYV_	Codes for symbol characters, shapes, and gestures.
TPD_	Options for **TrimPenData** function.
TPT_	Flags for **TARGINFO** structure.
VWM_	Flag values for **PostVirtualMouseEvent**.
WCR_	Configuration options for **ConfigHREC** and **ConfigRecognizer**.
WLT_	Word list types.

Pen Application Programming Interface Functions

This chapter provides a reference listing of the pen API functions, arranged in alphabetical order. Each entry describes a separate function organized under the following margin headings:

Margin heading	Description
Parameters	List of function parameters
Return Value	Possible return values and their meanings
Comments	Additional information about the function
See Also	Cross-reference to related API services

Next to each function name is a number that identifies the pen API version that supports the function – for example, 1.0 or 2.0.

The names of application callback functions appear in italics to indicate the names are placeholders. Callback functions can have any name.

Constants that pertain only to a specific function are listed in this chapter in the reference entry for that function. Generally, constants that pertain to two or more API services appear in Chapter 13, "Pen Application Programming Interface Constants."

AddInksetInterval Overview

Group

2.0

Merges an interval into an inkset.

BOOL AddInksetInterval(HINKSET *hinkset*, **LPINTERVAL** *lpiNew* **)**

Parameters

hinkset

Handle to an inkset.

lpiNew

Address of an **INTERVAL** structure.

Return Value

Returns TRUE if successful; otherwise FALSE.

Comments

The inkset is reallocated to a larger size by this function. The interval merges with any existing intervals, changing the interval only when required. For example, if the new interval is a subset of an existing one, there will be no change. Similarly, if the new interval overlaps an existing one, the union is formed. The maximum number of intervals allowed is defined to be (65536 - sizeof(INKSET)) / sizeof(INTERVAL), which evaluates to 5460.

The ending time of the new interval must be greater than or equal to the beginning time. If the interval has a duration of 0, **AddInksetInterval** does nothing, but returns TRUE.

An inkset formed using this function is guaranteed to have the intervals in ascending chronological order.

See Also

GetInksetInterval, **INTERVAL**

AddPenDataHRC Overview

Group

2.0

Adds an **HPENDATA** object to an **HRC** object for recognition.

int AddPenDataHRC(HRC *hrc*, **HPENDATA** *hpndt* **)**

Parameters

hrc

Handle to the **HRC** object.

hpndt

Handle to the **HPENDATA** object.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_INVALIDPNDT	Invalid pen data object.

Comments

Before terminating, the application must free the pen data, using **DestroyPenData**. Because the recognizer copies any data it requires, the recognizer does not affect the original data.

Calling this function is equivalent to adding data to the recognition context by walking the pen data strokes from beginning to end in stroke order. (Note that the stroke order may not necessarily be in chronological order if insertions have been made.)

A recognizer is not required to use or maintain OEM data; that is, a recognizer may choose to ignore some or all of the OEM data it receives from **AddPenDataHRC**. This means that the **HPENDATA** object that the recognizer returns through its **CreatePenDataHRC** function may differ from *hpndt* in its OEM data.

See Also

CreatePenData, **AddPenInputHRC**

AddPenInputHRC Overview

Group

2.0

Adds pen data to an **HRC** object for recognition. A recognizer must export this function.

int AddPenInputHRC(HRC *hrc*, **LPPOINT** *lppt*, **LPVOID** *lpvOem*, **UINT** *fuOem*, **LPSTROKEINFO** *lpsi* **)**

Parameters

hrc

Handle to the **HRC** object.

lppt

Address of an array of **POINT** structures.

lpvOem

Address of a buffer containing OEM data, or NULL if there is no OEM data.

fuOem

Flags to specify which OEM data is valid.

lpsi

Address of a **STROKEINFO** structure.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.

Comments

A recognizer is not required to use or maintain OEM data; that is, a recognizer may choose to ignore some or all of the OEM data it receives from **AddPenInputHRC**. This means that the **HPENDATA** object that the recognizer returns through its **CreatePenDataHRC** function may differ from *hpndt* in its OEM data.

See Also

GetPenInput, **GetPenDataStroke**

AddPointsPenData Overview

Group

1.0 2.0

Adds a set of data points to the pen data object.

HPENDATA AddPointsPenData(HPENDATA *hpendata*, **LPPOINT** *lppt*, **LPVOID** *lpvOem*, **LPSTROKEINFO** *lpsiNew* **)**

Parameters

hpendata

Handle to a pen data object.

lppt

Address of an array of **POINT** structures containing new data points to be added to the pen data. Zero points can be added to force a change of pen state or to set a new pen state.

lpvOem

OEM data. Can be set to NULL if there is no additional OEM data. The pen data header determines how the OEM data is interpreted.

lpsiNew

Address of a **STROKEINFO** structure for new stroke data. Contains the count of points from *lppt* to be added.

Return Value

Returns a handle to the pen data object. Normally, this is the same handle originally passed to the function. NULL is returned on error. The size of *hpendata* is limited to 64K.

Comments

A call to **GetPenHwEventData** or **GetPenInput** gets the *lpsiNew* and *lpvOem* values. A subsequent call to **AddPointsPenData** appends the set of points to the **HPENDATA** memory block identified by *hpendata*. The *lpsiNew* argument points to a **STROKEINFO** structure that describes the new points, and *lpvOem* points to the corresponding OEM data (if any) to be added along with the points.

The **STROKEINFO** structure indicates the pen state of the new points—that is, whether the pen is up or down. To avoid unnecessarily creating new strokes in the **HPENDATA** block, **AddPointsPenData** compares the pen state of the new points with the pen state of the last stroke in the **HPENDATA** block. If the new points have the same pen state as the last stroke, the function appends the points to the last stroke and updates the last **STROKEINFO** structure within the **HPENDATA** block. If the new points have a different pen state, **AddPointsPenData** appends them to the **HPENDATA** block as a new stroke, along with the **STROKEINFO** structure pointed to by *lpsiNew*.

AddPointsPenData does not scale the data points. The calling application must ensure that the added data points have the same scale as the rest of the **HPENDATA** block.

See Also

CreatePenData, **GetPenHwEventData**

AddWordsHWL Overview

Group

2.0

Adds words to a word list.

int AddWordsHWL(HWL *hwl*, **LPSTR** *lpsz*, **UINT** *uType* **)**

Parameters

hwl

Handle to a word list, or the constant HWL_SYSTEM for the recognizer's master word list.

lpsz

A pointer to a source of words, depending on the *uType* parameter.

uType

Word list type. This can be one of the following values:

Constant	Description
WLT_STRING	*lpsz* points to a single null-terminated character string in memory.
WLT_STRINGTABLE	*lpsz* points to an array of null-terminated character strings in memory. The list is terminated by two null characters.
WLT_WORDLIST	*lpsz* is the handle of a previously created word list, cast as **LPSTR**.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.

Comments

If a user wants to add a word to the system word list, which is available whenever the system dictionary is enabled (see **EnableSystemDictionaryHRC**), then *hwl* should be set to the predefined constant HWL_SYSTEM. Words that are not normally found in a dictionary, such as a person's name, can be added to the system word list. How this list is implemented, its size, or if it even exists, depends on the recognizer. A typical recognizer might maintain a thousand-word list, replacing random entries on overflow.

The HWL_SYSTEM constant cannot be used in any of the other word-list functions. For example, it is not possible to destroy the system word list with the **DestroyHWL** function.

For a description of word lists and how a recognizer uses them, see "Configuring the HRC" in Chapter 5, "The Recognition Process."

See Also

CreateHWL, EnableSystemDictionaryHRC

AnimateProc

2.0

The **AnimateProc** function is an application-defined callback function that provides information to **DrawPenDataEx** on a periodic basis. The name *AnimateProc* serves only as a placeholder; the function can have any name.

BOOL CALLBACK AnimateProc(HPENDATA *hpndt*, **UINT** *iStrk*, **UINT** *cPnt*, **UINT FAR *** *lpuSpeedPct*, **LPARAM** *lParam* **)**

Parameters

hpndt

Handle to the pen data currently being drawn.

iStrk

Zero-based index to the stroke being drawn, or about to be drawn.

cPnt

Count of points already drawn in this stroke.

lpuSpeedPct

Address of the speed-percent value.

lParam

Application-specific data passed to the callback. This value is specified in **ANIMATEINFO**.

Return Value

The callback function must return TRUE to continue drawing the pen data. Returning FALSE stops animation immediately.

Comments

One of the parameters of **DrawPenDataEx** provides the address of this callback function. The application must create an instance of this function using the **MakeProcInstance** function, and ensure that it is exported in the module-definition (.DEF) file.

The application can monitor the state of animation or provide the user with an opportunity to change the speed of animation, including pausing it, using the value addressed by *lpuSpeedPct*.

The application can also pass application-specific information to the callback in *lParam*. For example, a handle to the **DC** (device context) can be passed.

Callbacks are made at the beginning of the stroke or time interval, before any drawing is done. However, if AI_SKIPUPSTROKES is specified, a callback is not made before up strokes.

See Also

DrawPenDataEx, **ANIMATEINFO**

AtomicVirtualEvent Overview

Group

1.0 2.0

Locks out pen packets.

void AtomicVirtualEvent(BOOL *fBegin* **)**

Parameters

fBegin

Flag for beginning or ending lockout. TRUE begins lockout, FALSE ends it.

Return Value

This function does not return a value.

Comments

AtomicVirtualEvent is used by the Pen Palette or a similar virtual-keyboard program to lock out pen packets while the application is posting simulated key or mouse events.

Calling **AtomicVirtualEvent** with a TRUE value blocks input from physical devices until they are freed with a call specifying FALSE. Applications should end the lockout as quickly as possible.

An interruptable thread should not call **AtomicVirtualEvent**.

Example

The following code fragment posts a mouse click:

```
AtomicVirtualEvent( TRUE );
PostVirtualMouseEvent( VWM_MOUSELEFTDOWN, xPos, yPos );
PostVirtualMouseEvent( VWM_MOUSEMOVE, xPos, yPos );
PostVirtualMouseEvent( VWM_MOUSELEFTUP, xPos, yPos );
AtomicVirtualEvent( FALSE );
```

See Also

PostVirtualKeyEvent, **PostVirtualMouseEvent**

BeginEnumStrokes Overview

Group

1.0 2.0

Locks a pen data block in memory in preparation for enumerating strokes.

Note This function is provided only for compatibility with version 1.0 of the Pen API, and will not be supported in future versions.

LPPENDATA BeginEnumStrokes(HPENDATA *hpendata* **)**

Parameters

hpendata

Handle to an **HPENDATA** object.

Return Value

Returns a pointer to the locked pen data if successful. Returns NULL if *hpendata* is compressed or if the handle cannot be locked.

Comments

BeginEnumStrokes calls the **GlobalLock** function internally, returning a far pointer to the memory block in the global heap. This serves to lock the data in preparation for direct reading or calling **GetPenDataStroke**. The return value from **BeginEnumStrokes** is used as an argument for **GetPenDataStroke**. After calling **BeginEnumStrokes** to lock data, an application must unlock the data when finished by calling **EndEnumStrokes**.

An application should never modify data directly within an **HPENDATA** block. Doing so can invalidate other information in the block. To modify an **HPENDATA** block, use one of the Pen API functions listed in Chapter 4, "The Inking Process."

See Also

EndEnumStrokes, **GetPenDataStroke**

BoundingRectFromPoints Overview

Group

1.0 2.0

Calculates a rectangle that bounds a range of points.

void BoundingRectFromPoints(LPPOINT *lppt*, **UINT** *cPt*, **LPRECT** *lprect* **)**

Parameters

lppt

Address of an array of **POINT** structures.

cPt

Number of **POINT** structures in the array. This parameter can be 0.

lprect

Address of a **RECT** structure that contains the bounding rectangle when the function returns.

Return Value

This function does not return a value.

Comments

The bounding rectangle is empty at [0,0] if there are no points. For a single point, the rectangle is empty at that point.

CharacterToSymbol Overview

Group

1.0 2.0

Converts an ANSI string to an array of SYV_ symbol values.

int CharacterToSymbol(LPSTR *lpstr*, **int** *cSyv*, **LPSYV** *lpsyv* **)**

Parameters

lpstr

Address of a null-terminated ANSI string to be converted.

cSyv

Maximum number of SYV_ symbols the array *lpsyv* can hold.

lpsyv

Address of an array of SYV_ symbol values into which **CharacterToSymbol** places the converted symbols. The array must be large enough to hold *cSyv* symbols.

Return Value

Returns the number of characters converted, or -1 if there is an error.

Comments

Conversion proceeds until a null byte is found in *lpstr* or until *lpsyv* has been filled with *cSyv* symbols. A null byte is converted to SYV_NULL.

See Also

SymbolToCharacter, **SYG**, SYV_

CompactPenData Overview

Group

1.0 2.0

Compacts pen data based on specified trim options.

Note This function is provided for compatibility with version 1.0 of the Pen API and will not be supported in future versions. Use **TrimPenData** and **CompressPenData** instead.

HPENDATA CompactPenData(HPENDATA *hpndt*, **UINT** *fuTrim* **)**

Parameters

hpndt

Handle to a pen data object.

fuTrim

Data-trimming options:

PDTT_DEFAULT

Reallocates memory block to fit the data; does not trim the data. If you call **CompactPenData** with this trim option and then call the **GlobalSize** function with the pen data handle as a parameter, you can retrieve the size of the pen data.

PDTT_ALL

Removes the **PENINFO** structure from the header. Discards all data from pen-up points (points collected when the pen is not in contact with the tablet), and removes OEM data and collinear points.

PDTT_COLLINEAR

Removes successive identical (coincident) points and collinear points from the pen data. After the operation is performed, PDTS_NOCOLLINEAR is set in the **wPndts** member of the **PENDATAHEADER** structure. The collinear points can be removed with very little if any loss of recognition accuracy. If the collinear points are removed before the points are scaled to display coordinates, there may be a small change in the displayed image.

PDTT_COMPRESS

Compresses the data without losing any information. After the data has been compressed, the compressed handle to the pen data can be passed as a parameter only to the functions **CompactPenData**, **GetPenDataInfo**, and **DuplicatePenData**. **CompactPenData** uses a "lossless" compression method that retains the ability for an application to recognize the ink after subsequent decompression. You can use this option with other trim options, including PDTT_DECOMPRESS. In this case, compression is done after all other options have been satisfied.

PDTT_DECOMPRESS

Decompresses the data. You can use this option with other trim options, including PDTT_COMPRESS. In this case, decompression is performed first, followed by any other trim options specified, and followed by recompression if PDTT_COMPRESS is specified. Since the compression method used by **CompactPenData** does not lose information, the data is completely restored.

PDTT_OEMDATA

Removes all OEM data—this is data other than coordinates, such as pressure. This option does not affect delayed recognition unless a recognizer is being used that expressly requires OEM data. For example, signature recognizers often use pressure information.

PDTT_PENINFO

Removes the **PENINFO** structure from the header. You can use this option if there is no OEM data associated with the data points or if the application does not use any of the OEM data. This option has no effect on the pen data for delayed recognition. Any OEM data present is also removed.

PDTT_UPPOINTS

Removes all data from pen-up points (points collected when the pen is not in contact with the tablet). This option has no effect on delayed recognition. This option is not usually necessary because pen-up points are not a part of standard pen data.

Return Value

If successful, **CompactPenData** returns a handle to a pen data object; otherwise, it returns NULL. **CompactPenData** may fail and return NULL in low-memory situations if compression or decompression is requested.

Comments

The PDTS_ bits are set in the **wPndts** member of the **PENDATAHEADER** structure to indicate which operations have been performed.

See Also

CompressPenData, **TrimPenData**, **CreatePenData**, **PENINFO**, **PENDATAHEADER**

CompressPenData Overview

Group

2.0

Compresses or decompresses the data in an **HPENDATA** object.

int CompressPenData(HPENDATA *hpndt*, **UINT** *fuFlags*, **DWORD** *dwReserved* **)**

Parameters

hpndt

Handle to the **HPENDATA** object.

fuFlags

Specifies whether to compress or decompress the data, as follows:

Constant	Description
CMPD_COMPRESS	Compress the pen data.
CMPD_DECOMPRESS	Decompress the pen data.

dwReserved

Must be 0.

Return Value

This function returns one of the following:

Constant	Description
PDR_OK	Successful completion. Redundant operations, such as compressing an **HPENDATA** object that has already been compressed, are not errors.
PDR_ERROR	Illegal parameter or other error.
PDR_MEMERR	Memory error.
PDR_PNDTERR	Invalid pen data.
PDR_VERSIONERR	Could not convert old pen data.

Comments

This function replaces the the version 1.0 Pen API function **CompactPenData**, which is supported for compatibility only.

For a discussion of data compression, see "Compressing Pen Data" in Chapter 4, "The Inking Process."

See Also

CompactPenData, **TrimPenData**

ConfigHREC Overview

Group

2.0

Allows an application to set or query recognizer-specific values. All calls to **ConfigHREC** are serviced by the recognizer's **ConfigRecognizer** function. In version 2.0 of the Pen API, applications must call **ConfigHREC** rather than **ConfigRecognizer**.

int ConfigHREC(HREC *hrec*, **UINT** *uSubFunction*, **WPARAM** *wParam*, **LPARAM** *lParam* **)**

Parameters

hrec

Module handle of the recognizer library. If this value is NULL, the system default recognizer is used.

uSubFunction

Recognizer subfunction identifier. See the "Comments" section below.

wParam

Depends on the value of *uSubFunction*.

lParam

Address of a buffer. The contents of the buffer depend on the value of *uSubFunction*.

Return Value

If successful, returns 0 or a positive value as described in the list of *uSubfunction* constants below; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Missing recognizer, invalid parameter, or other error.
HRCR_MEMERR	Insufficient memory.

Comments

The *uSubFunction* parameter contains one of the following WCR_ values that identifies the requested configuration service:

WCR_CONFIGDIALOG

Instructs the recognizer to open a dialog box to set any recognizer-specific parameters. (This is analogous to DEVMODE in printer drivers, which is called when a user sets up a printer.) Some examples of the kind of settings a recognizer might implement are whether or not to allow cursive input, how much to depend on stroke order, and how rapidly to modify prototypes based on the user's own style.

The *lParam* parameter points to the name of the user currently selected in the Control Panel application. The *wParam* parameter is used by the recognizer as the parent window for any dialog boxes it displays. The return value is always TRUE.

WCR_DEFAULT

Returns TRUE if the recognizer is capable of being a default recognizer. A default recognizer must support the standard character set as well as standard gestures.

WCR_GETALCPRIORITY

Returns the current default alphabet priority being used by the recognizer. The *lParam* parameter points to a variable that specifies the alphabet priority as a bitwise-OR combination of ALC_ values. The *wParam* parameter is not used. This subfunction is used by the system; applications should instead get alphabet priority by calling **GetAlphabetPriorityHRC**. The return value is TRUE if successful.

WCR_GETANSISTATE

Returns TRUE if the recognizer can recognize all of the ANSI character set; otherwise, returns FALSE or HRCR_ERROR.

WCR_GETDIRECTION

If successful, returns the current writing direction assumed by the recognizer; otherwise, returns HRCR_ERROR.

WCR_GETHAND

If successful, returns 0 if the user writes with the right hand or nonzero if the user writes with the left hand; otherwise, returns HRCR_ERROR.

WCR_PRIVATE

Values above WCR_PRIVATE have a meaning dependent on the recognizer.

WCR_PWVERSION

Returns the version number of the Pen API for which this recognizer was created. This value is 2 for the current version.

WCR_QUERY

Returns TRUE if the recognizer supports a configuration dialog box.

WCR_QUERYLANGUAGE

The *wParam* parameter is not used. The *lParam* parameter points to a null-terminated language string. The return value is TRUE if the recognizer supports the language; otherwise, it is FALSE.

WCR_RECOGNAME

Retrieves an identification string from the recognizer. The *lParam* parameter is treated as a far pointer to a buffer that is filled with an identification string from the recognizer. The *wParam* parameter is the size of the buffer to fill. The identification string is a short description of the recognizer that Control Panel presents to the user. A sample string is "US English character set, cursive & print." The return value is always 0.

WCR_SETALCPRIORITY

Sets the current default alphabet priority for the recognizer to the value in *lParam*. Note that setting a priority for individual characters is not supported for defaults. The *wParam* parameter is not used. This subfunction is used by the system; applications should set priority explicitly in an **HRC** with the **SetAlphabetPriorityHRC** function. The return value is TRUE if successful.

WCR_SETANSISTATE

Sets a flag to enable or disable recognition of the entire ANSI character set. Setting *lParam* to 1 enables recognition of the entire ANSI set; setting *lParam* to 0 allows recognition of only English (ASCII) characters. The *wParam* parameter is not used.

The WCR_SETANSISTATE subfunction determines the default setting when **CreateCompatibleHRC** creates an **HRC**. An application can explicitly override the setting for the **HRC** with the **SetInternationalHRC** function. The return value is TRUE if successful.

WCR_SETDIRECTION

Sets the current writing direction for the recognizer to the value in *lParam*, which can be an

appropriate combination of the RCD_ values. The *wParam* parameter is not used. The return value is TRUE if successful.

WCR_SETHAND

Sets the current writing hand preference for the recognizer. The *lParam* parameter is 0 for a right-handed user or 1 for a left-handed user. The *wParam* parameter is not used. The return value is TRUE if successful.

WCR_TRAIN

This subfunction returns TRAIN_NONE if the recognizer does not support training. A return value of TRAIN_DEFAULT indicates support for the default trainer, including the capability of resetting its database to the original "factory" setting (see WCR_TRAINSAVE). A return value of TRAIN_CUSTOM indicates that the recognizer also provides its own custom trainer. A return value of TRAIN_BOTH indicates support for both kinds of training.

WCR_TRAINMAX

The recognizer returns the maximum number of SYV_ symbol values that it can train for any given shape.

The recognizer should return 0 if it can train any number of characters. For example, the Microsoft recognizer can train one character for a shape; a cursive recognizer may allow more.

WCR_TRAINSAVE

The trainer calls the **ConfigHREC** function with the parameters set to (WCR_TRAINSAVE, TRAIN_SAVE, 0) when it is time to save the data-base. This happens when the user closes the trainer. After this call, the recognizer should return TRUE if it can successfully save the database; otherwise, it should return FALSE.

The trainer calls the function with (WCR_TRAINSAVE, TRAIN_REVERT, 0) before it discards any changes made to the database that have not yet been saved to disk (that is, revert to saved). This happens when the user cancels the changes. The recognizer should return TRUE if it is successful.

The trainer can alternatively call **ConfigHREC** with (WCR_TRAINSAVE, TRAIN_RESET, 0) to reset the database to the original "factory" settings. The recognizer should return TRUE if it is successful.

WCR_TRAINDIRTY

The recognizer returns TRUE if the recognizer needs to save training. The recognizer returns FALSE if no training occurred, if the recognizer does not use a database for training, if the recognizer saves as it works, or if the recognizer cannot revert the training.

The *hwnd* parameter is a handle to the requesting window. The trainer can use this as the parent window for a dialog box, for example. If there has been a recent recognition, a pointer to it is passed in the *lParam* parameter, although this may be NULL.

The format for the WCR_TRAINDIRTY subfunction call is:

```
ConfigHREC( hrec, WCR_TRAINDIRTY, 0, 0 );
```

WCR_TRAINCUSTOM

If the recognizer returns TRAIN_CUSTOM or TRAIN_BOTH in response to WCR_TRAIN, it will receive a WCR_TRAINCUSTOM message when it is time to display its own training system.

The format for the WCR_TRAINCUSTOM subfunction call is:

```
ConfigHREC( hrec, WCR_TRAINCUSTOM, hwnd, lprcresult );
```

WCR_USERCHANGE

Notifies the recognizer of a change in user. The *lParam* parameter points to a null-terminated string

containing the user's name. The *wParam* parameter specifies the required modification:

A *wParam* value of CRUC_NOTIFY indicates a new user, the name of whom is in the string that *lParam* points to.

A *wParam* value of CRUC_REMOVE indicates that the user identified by *lParam* should be removed from the recognizer's user list. If the recognizer has saved any files or settings for the user, they should be deleted in response to this notification.

WCR_VERSION

Returns the version number. The low-order byte of the return value specifies the major (version) number. The high-order byte specifies the minor (revision) number.

See Also

ConfigRecognizer, ALC_, SYV_

ConfigRecognizer Overview

Group

1.0 2.0

Provides system access to the configuration settings of a recognizer. In version 2.0 of the Pen API, only the pen system can call **ConfigRecognizer**. Applications must call **ConfigHREC** to query or set recognizer configuration values. The system routes **ConfigHREC** calls to the **ConfigRecognizer** function of the appropriate recognizer.

A recognizer must export **ConfigRecognizer**. The information in this entry is for recognizer developers only, not application developers.

UINT ConfigRecognizer(UINT *uSubFunction*, **WPARAM** *wParam*, **LPARAM** *lParam* **)**

Parameters

uSubFunction

Recognizer subfunction identifier. See **ConfigHREC** for descriptions of the WCR_ subfunctions that **ConfigRecognizer** must support. In addition, **ConfigRecognizer** must support the following two WCR_ subfunctions:

WCR_INITRECOGNIZER

When an application installs a recognizer by using **InstallRecognizer**, the system calls the recognizer's **ConfigRecognizer** function with the WCR_INITRECOGNIZER subfunction. The *wParam* parameter is not used and *lParam* is a far pointer to an ASCII string containing the user's name, as set in the system registry. If successful, the recognizer should return 1; otherwise, it should return 0 to indicate an error.

In response to the WCR_INITRECOGNIZER subfunction, the recognizer should perform any required initialization tasks. (This subfunction replaces the **InitRecognizer** function exported by version 1.0 recognizers.)

WCR_CLOSERECOGNIZER

When an application unloads a recognizer by using **UninstallRecognizer**, the system calls the recognizer's **ConfigRecognizer** function with the WCR_CLOSERECOGNIZER subfunction. The *wParam* and *lParam* parameters are not used. If successful, the recognizer should return 1; otherwise, it should return 0 to indicate an error.

In response to the WCR_CLOSERECOGNIZER subfunction, the recognizer should perform any required cleanup tasks. (This subfunction replaces the **CloseRecognizer** function exported by version 1.0 recognizers.)

wParam

Depends on the value of *uSubFunction*.

lParam

A value, or an address of a buffer. The contents of the buffer depend on the value of *uSubFunction*.

Return Value

Returns 0 or a positive value, depending on *uSubFunction*.

Comments

ConfigRecognizer provides initialization and query services for the pen system. The parameter *uSubFunction* is a WCR_ value that specifies the configuration service that **ConfigRecognizer** must perform.

When an application calls **ConfigHREC**, the system determines the appropriate recognizer and passes the call to that recognizer's **ConfigRecognizer** function. **ConfigHREC** exists only because its extra argument *hrec* identifies to the system the intended recognizer library. This information is necessary in version 2.0 of the Pen API, which allows multiple recognizer libraries to exist simultaneously. Thus, the names **ConfigHREC** and **ConfigRecognizer** refer to the same function. Applications refer to the function as **ConfigHREC**, while recognizers export it as **ConfigRecognizer**.

See Also

ConfigHREC, SYV_

CorrectWriting Overview

Group

1.0 2.0

Sends text to the CorrectText dialog box to allow the user to edit text using a single-line or multiline bedit control.

BOOL CorrectWriting(HWND *hwnd*, **LPSTR** *lpText0*, **UINT** *cbText0*, **LPVOID** *lpvReserved*, **DWORD** *dwFlags*, **DWORD** *dwParam* **)**

Parameters

hwnd

Handle of the owner of the CorrectText dialog box or writing tools used to edit the text.

lpText0

Far pointer to a buffer containing the text to be corrected.

When **CorrectWriting** returns, the *lpText0* buffer holds the corrected text. As a general rule, this parameter should allow for growth by a factor of at least two or some maximum size that depends on the field of entry.

cbText0

Number of characters in *lpText0*. This value must be greater than 1 and include a byte for the string's null terminator.

lpvReserved

This parameter is reserved and should be set to NULL.

dwFlags

Translation and style flags, formed by the low-order word and high-order word of *dwFlags*. The low-order word must be one or more of the following flags, combined with the bitwise-OR operator. Note that the CWR_REPLACECR and CWR_REPLACETAB flags replace CWR_STRIPTAB and CWR_STRIPCR, respectively; both flags are in version 1.0 of the Pen API.

Constant	Description
CWR_BOXES	Create bedit writing tool instead of keyboard. This flag can be used only for edit control and its derivatives. Use of this flag by applications is not recommended.
CWR_HEDIT	Indicates that the given *hwnd* is an edit control or a control derived from the edit control. This flag can be used only for edit control and its derivatives. Use of this flag by applications is not recommended.
CWR_INSERT	Use "Insert Text" instead of "Edit Text" as the title. CWR_TITLE overrides this flag.
CWR_KEYBOARD	Create keyboard writing tool instead of bedit lens. This flag can be used only for edit control and its derivatives. Use of this flag by applications is not recommended.
CWR_KKCONVERT	Initiate IME (Japanese version only).
CWR_REPLACECR	Replace carriage return characters in the text in the buffer by spaces just before the

	call returns.
CWR_REPLACETAB	Replace tabs in the text in the buffer by spaces just before the call returns.
CWR_SIMPLE	Use writing tool (simple dialog box). This flag can be used only for an edit control and its derivatives. Use of this flag by applications is not recommended.
CWR_SINGLELINEEDIT	Replace carriage returns and tabs with spaces and strip linefeeds from the text in the buffer just before the call returns.
CWR_STRIPLF	Strip linefeed characters from the text in the buffer just before the call returns.
CWR_TITLE	Interpret *dwParam* (see below) as a pointer to the title text string.

The high-order word must be one of the following values and cannot be combined with the bitwise-OR operator. The values determine the type of keyboard to show when the user clicks the keyboard button in the dialog box.

Constant	Description
CWRK_TELPAD	Use the telephone-type keyboard.
CWRK_BASIC	Use the basic keyboard.
CWRK_DEFAULT	Use the default keyboard type. The default keyboard type is currently the same as the basic keyboard type.
CWRK_FULL	Use the full keyboard.
CWRK_NUMPAD	Use the numeric keyboard.

dwParam

A far pointer to a text string that serves as the title of the dialog box if CWR_TITLE is present in *dwFlags*; otherwise, this parameter must be 0.

Return Value

Returns TRUE if the writing tool or **CorrectWriting** operation was successful. Otherwise, the return value is FALSE.

Comments

CorrectWriting sends a WM_PENMISC message with PMSC_GETHRC as the *lParam* to the specified window. This message requests the **HRC** handle associated with the window, which the system then uses for the dialog box. The window should return a copy of its **HRC** so that the system can destroy it before the call returns. If the window returns NULL to this message, the system creates a default **HRC**.

Note that in the Japanese version, **CorrectWriting** is supported but internally calls **CorrectWritingEx**, which opens a Dialog Input Window.

CorrectWritingEx Overview

Group

2.0

Sends text to the CorrectText dialog box to allow the user to edit text using the Japanese Data Input Window. (Japanese version only.)

INT CorrectWritingEx(HWND *hwnd*, **LPSTR** *lpText*, **UINT** *cbText*, **LPCWX** *lpcwx* **)**

Parameters

hwnd

Handle of the owner of the CorrectText dialog box or writing tool used to edit the text. This can be NULL.

lpText

Far pointer to a buffer containing text to correct. This is copied into the Data Input Window's edit control. If *lpText* is NULL, a WM_GETTEXT message is sent to the text source window, specified by the *hwndText* member of *lpcwx*, or if *lpcwx* or its *hwndText* member is NULL, to *hwnd*. On successful exit, a WM_SETTEXT message will be sent to that window with modified text.

cbText

Size of the *lpText* buffer. If the source of the text is an edit control constrained by EM_LIMITTEXT, *cbText* should reflect that size. If *lpText* is NULL, the *cbText* value will be used to limit text if it is greater than zero; otherwise, no limit is used and the returned text may be of arbitrary size.

lpcwx

Address of a **CWX** structure, or NULL. The structure is used to specify optional correction parameters; for a description of its members, see **CWX**. If this value is NULL, the following default assumptions are made:

- The text window is the same as the owner window *hwnd.*
- A default recognition context is used.
- The edit control style is a combination of ES_LEFT and ES_MULTILINE.
- All text is selected; the caption is "Edit Text".
- Most recently use values for context flags, keyboard, keyboard states, position, and size are used.

Return Value

If there is a programming or memory error, the negative value CWXR_ERROR is returned. Otherwise, one of the following nonnegative values is returned:

Constant	Description
CWXR_MODIFIED	User pressed the OK button.
CWXR_UNMODIFIED	User pressed the Cancel button, or closed the dialog, or pressed the OK button but did not make any changes to the text.

Comments

An application must be sure to initialize the **CWX** structure properly if it is used. In particular, the **cbSize** member must be set to sizeof(CWX), and the remaining fields (at least up to **dwSel**) are typically set to zero.

Example

The following example shows how to initialize and call **CorrectWritingEx** when a button is pressed in a dialog:

```
CWX cwx = {sizeof(CWX), 0, NULL, NULL, {0}, 0L, 0L};

cwx.hwndText = GetDlgItem(hdlg, IDD_ETSL);     // dialog edit
cwx.dwEditStyle = GetWindowLong(cwx.hwndText, GWL_STYLE)
     | ES_PASSWORD;
cwx.dwSel = SendMessage(cwx.hwndText, EM_GETSEL, 0, 0);
_fstrcpy((LPSTR)cwx.szCaption, (LPSTR)"Enter your password:");

// we specify kbd and context, but use MRU placement
cwx.wApplyFlags = CWXA_KBD | CWXA_STATE | CWXA_CONTEXT;

// don't update most-recently used settings for this one-shot:
cwx.wApplyFlags |= CWXA_NOUPDATEMRU;
cwx.ixkb = CWXK_QWERTY;
cwx.rgState[CWXK_QWERTY-CWXK_FIRST] = CWXKS_HAN | CWXKS_ROMA;
cwx.dwFlags = CWX_NOTOOLTIPS | CWX_TOPMOST;    // no distractions

if (CorrectWritingEx(hdlg, NULL, 0, &cwx) != CWXR_MODIFIED)
     ErrBox(EB_WHOAREYOU);
// validate pwd in the text window etc...
```

See Also

CWX

CreateCompatibleHRC Overview

Group

2.0

Creates a handwriting recognition context **HRC** that can be used to do handwriting recognition, optionally compatible with an existing context template. A recognizer must export this function.

HRC CreateCompatibleHRC(HRC *hrcTemplate***, HREC** *hrec* **)**

Parameters

hrcTemplate

Handle to an existing **HRC** object that can provide default settings for the recognition context being created. If NULL, this parameter is ignored and default settings are used.

hrec

Instance handle of the recognizer library. This is the value returned by the Windows function **LoadLibrary**. Note that the module handle returned by the Windows function **GetModuleHandle** does not work in this case. If this value is NULL, the system default recognizer is used by internally making a call to **GetPenMiscInfo** with PMI_SYSREC as the first argument.

Return Value

Returns a handle to a new **HRC** object if successful; otherwise, returns NULL.

Comments

The *hrcTemplate* parameter can be used to copy an old context into the new **HRC** object. This includes settings such as word lists, coercion, and **GUIDE** structure, but excludes any pen data that may be in the old context.

See Also

DestroyHRC, **GetResultsHRC**, **SetMaxResultsHRC**

CreateHWL Overview

Group

2.0

Creates a handle to a word list.

HWL CreateHWL(HREC *hrec*, **LPSTR** *lpsz*, **UINT** *uType*, **DWORD** *dwReserved* **)**

Parameters

hrec

Module handle of the recognizer library. If this value is NULL, the system default recognizer is used.

lpsz

A pointer to a source of words, depending on the *uType* parameter.

Type

Word-list type. This can be one of the following values:

Constant	Description
WLT_EMPTY	An empty word list is created. The *lpsz* parameter is ignored.
WLT_STRING	The *lpsz* parameter points to a single null-terminated character string in memory.
WLT_STRINGTABLE	The *lpsz* parameter points to an array of null-terminated character strings in memory. The list is terminated by two null characters.

dwReserved

Must be 0.

Return Value

If successful, returns the handle of a newly created word list; otherwise, returns NULL. If the recognizer does not support word lists, the return value is NULL.

Comments

CreateHWL creates a word list for constraining recognition. Word lists can be combined using the **AddWordsHWL** function.

To make a word list from words in a file, an application uses **CreateHWL** to create an empty word list, then reads the file into it with the **ReadHWL** function.

Any word lists created by an application must eventually be destroyed by calling **DestroyHWL**. Attempting to unload a recognizer that has open word lists results in an error.

For a description of word lists and how a recognizer uses them, see "Configuring the HRC" in Chapter 5, "The Recognition Process."

Example

The following example demonstrates how to provide a word list to constrain recognition results to the words "Canada," "USA," or "Mexico":

```
static char szNames[] = { "Canada",
                          "USA",
                          "Mexico"
                        };

HWL hwlCountries = CreateHWL( NULL,
     (LPSTR)szNames,
     WLT_STRINGS, 0L );            // Create early for later use
     .
     .
     .
if (hrc = CreateCompatibleHRC( NULL, NULL ))
{
   SetWordlistHRC( hrc, hwlCountries );       // Set list into HRC
   SetWordlistCoercionHRC( hrc, SCH_FORCE );  // Force match
        .
        .     // Code that collects and recognizes input goes here
        .
}
```

See Also

AddWordsHWL, DestroyHWL, SetWordlistHRC

CreateInkset Overview

Group

2.0

Creates an empty inkset.

HINKSET CreateInkset(UINT *gmemFlags* **)**

Parameters

gmemFlags

Flag that specifies whether or not the Windows **GlobalAlloc** function should create a shared memory object when the inkset object is created. This flag should be either 0 or GMEM_DDESHARE. The GMEM_MOVEABLE and GMEM_ZEROINIT flags are added to this value, and other GMEM_ flags are ignored.

Return Value

Returns a handle to an inkset if successful; otherwise, the return value is NULL.

See Also

DestroyInkset, **INTERVAL**

CreateInksetHRCRESULT Overview

Group

2.0

Creates an inkset from parts of a recognition result.

HINKSET CreateInksetHRCRESULT(HRCRESULT *hrcresult*, **UINT** *iSyv*, **UINT** *cSyv* **)**

Parameters

hrcresult

Handle of an **HRCRESULT** object.

iSyv

Index to first symbol for inkset.

cSyv

Count of symbols.

Return Value

Returns the handle of a newly created inkset if successful. If the index to the first symbol *iSyv* is invalid, or some other error occurs, the return value is NULL.

Comments

The inkset spans a series of continuous symbols; disjoint sets are not allowed. Before terminating, the calling application must destroy the **HINKSET** object by calling **DestroyInkset**.

If the range of symbols specified by *iSyv* + *cSyv* exceeds the number of symbols available, the returned inkset is valid only for available symbols. This is not an error, so it is possible to assign *cSyv* a large value to get an inkset for all symbols after *iSyv*.

For a description of inksets, see "The HINKSET Object" in Chapter 4, "The Inking Process."

See Also

DestroyInkset, **GetResultsHRC**

CreatePenData Overview

Overview

1.0 2.0

Creates an empty **HPENDATA** block.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions. Use **CreatePenDataEx** instead.

HPENDATA CreatePenData(LPPENINFO *lppeninfo*, **int** *cbOem*, **UINT** *uScale*, **UINT** *gmemFlags* **)**

Parameters

lppeninfo

Address of tablet information to be inserted into the **PENINFO** structure in the pen data header. If this parameter is NULL, the current tablet settings are retrieved from the hardware instead. If there is no tablet, the pen data will not have an embedded **PENINFO** section and the **wPndts** member in **PENDATAHEADER** will have the PDTS_NOPENINFO flag set.

cbOem

Width of OEM data packet. If this value is greater than or equal to 0, the OEM data overrides the contents of the **PENINFO** structure, if present; otherwise, a negative value such as -1 can be used to specify that the system should calculate the size of the OEM data packet.

uScale

Data-scaling metric value. This parameter can be one of the following values:

Constant	Description
PDTS_LOMETRIC	Each logical unit is mapped to 0.1 millimeter. Positive x is to the right; positive y is down.
PDTS_HIMETRIC	Each logical unit is mapped to 0.01 millimeter. Positive x is to the right; positive y is down.
PDTS_HIENGLISH	Each logical unit is mapped to 0.001 inch. Positive x is to the right; positive y is down.
PDTS_ARBITRARY	The application has done its own scaling of the data point.
PDTS_STANDARDSCALE	The standard scaling metric; equivalent to PDTS_HIENGLISH.

gmemFlags

Flag that specifies whether or not the Windows **GlobalAlloc** function should create a shared memory object when the pen data object is created. This should be either 0 or GMEM_DDESHARE. The GMEM_MOVEABLE and GMEM_ZEROINIT flags are added to this value, and other GMEM_ flags are ignored.

Return Value

Returns a handle to a new and empty pen data object if successful; otherwise, it returns NULL.

Comments

The application provides the **PENINFO** structure for the header, the real size of any OEM data stored with each coordinate, and the scale of the coordinates.

The *uScale* parameter specifies scaling values that are also used in the **MetricScalePenData** function and in the **PENDATAHEADER** structure member **wPndts**. The scaling values do not behave in the same way as the Windows scaling units with similar names. For example, a 1-inch line in MM_HIENGLISH will not necessarily be an inch long on the screen because GDI does not know the size of the monitor. However, with PDTS_HIENGLISH in **MetricScalePenData**, a line drawn an inch long is actually an inch long.

If *lppeninfo* is NULL, and if there is no tablet on the system (that is, if **SendDriverMessage** fails), it returns NULL.

The *cbOem* value must be less than or equal to 12, depending on the size of the OEM data packet. A value of 0 explicitly sets the amount of OEM information to none. A negative value indicates that the size of the OEM data packet is to be calculated by the system. Any existing value for the **cbOemData** member of **PENINFO** can be overwritten.

See Also

CreatePenDataEx, **DestroyPenData**, PDTS_

CreatePenDataEx Overview

Overview

2.0

Creates a **PENDATA** structure with specified OEM data subsets.

HPENDATA CreatePenDataEx(LPPENINFO *lppeninfo*, **UINT** *uScale*, **UINT** *fuOptions*, **UINT** *gmemFlags* **)**

Parameters

lppeninfo

Address of tablet information to be inserted into the **PENINFO** structure in the pen data header. If this parameter is NULL, the current tablet settings are retrieved from the hardware instead. If there is no tablet, the pendata will not have an embedded **PENINFO** section and the **wPndts** member in **PENDATAHEADER** will have the PDTS_NOPENINFO flag set.

uScale

Data-scaling metric value. This parameter can be one of the following values:

Constant	Description
PDTS_LOMETRIC	Each logical unit is mapped to 0.1 millimeter. Positive x is to the right; positive y is down.
PDTS_HIMETRIC	Each logical unit is mapped to 0.01 millimeter. Positive x is to the right; positive y is down.
PDTS_HIENGLISH	Each logical unit is mapped to 0.001 inch. Positive x is to the right; positive y is down.
PDTS_ARBITRARY	The application has done its own scaling of the data point.
PDTS_STANDARDSCALE	The standard scaling metric is equivalent to PDTS_HIENGLISH.

fuOptions

Storage and trim options. If this parameter is 0, no timing, PDK_, or OEM data is stored. If it is CPD_DEFAULT, everything but user data is stored.

Otherwise, this parameter can explicitly specify subsets of OEM and other data. To do so, the parameter should be a combination of one of the CPD_USER values that allocate extra storage and any collection of PHW_ constants. (These values should be combined using the bitwise-OR operator.)

The following table lists the PHW_ values for the *fuOptions* parameter:

Constant	Description
PHW_PRESSURE	Report pressure in OEM data if available.
PHW_HEIGHT	Report height in OEM data if available.
PHW_ANGLEXY	Report XY-angle in OEM data if available.
PHW_ANGLEZ	Report Z-angle in OEM data if available.
PHW_BARRELROTATION	Report barrel rotation in OEM data if available.
PHW_OEMSPECIFIC	Report OEM-specific value in OEM data if

	available.
PHW_PDK	Report per-point PDK_ bits in OEM data.
PHW_ALL	Report all available OEM data. This flag is the sum of all other PHW_ flags.
	The following table lists the CPD_ values for the *fuOptions* parameter:
CPD_DEFAULT	Store timing, PDK, and all OEM data for each stroke.
CPD_USERBYTE	Set internal flag to add space for one byte of additional storage to be allocated for each stroke. Added space is for application use.
CPD_USERWORD	Set internal flag to add space for one word of additional storage to be allocated for each stroke. Added space is for application use.
CPD_USERDWORD	Set internal flag to add space for one doubleword of additional storage to be allocated for each stroke. Added space is for application use.
CPD_TIME	Maintain absolute time information for each stroke.

gmemFlags

Flag that specifies whether **GlobalAlloc** should create a shared memory object or not when the pen data object is created. This should be either 0 or GMEM_DDESHARE. The GMEM_MOVEABLE and GMEM_ZEROINIT flags are added to this value, and other GMEM_ flags are ignored.

Return Value

Returns the handle to the **HPENDATA** object if successful; otherwise, returns NULL.

Comments

CreatePenDataEx is an extension of **CreatePenData** that allows a more detailed specification of what is stored in each stroke of the pen data.

The *fuOptions* parameter is typically specified as CPD_DEFAULT to request collection and storage of all information generated by the tablet, including x-y data, absolute stroke timing information, and all available OEM data. The OEM data set that is actually stored in the pen data is the minimum set that satisfies both the request and what is physically available from the tablet (that is, intersection set).

If *lppeninfo* is NULL, and if there is no tablet on the system (that is, if the **SendDriverMessage** function fails), the pen data that is created will not have any hardware or OEM information and a default sampling rate of 100Hz will be used. This case is similar to removing **PENINFO** from the header using **TrimPenData** with a parameter of TPD_PENINFO.

A value of 0 for *fuOptions* is used to indicate that only coordinate data is required. While recognition of this type of pen data may suffer, this provides the least complicated type of pen data.

PHW_ bits can be specified to indicate which OEM values or per-point PDK_ pen state information is to be collected. Note that, except for PHW_PDK, which is always valid, this is only a request; if the hardware does not support certain types of OEM data, that data will be absent.

The *uScale* parameter specifies scaling values that are also used in the **MetricScalePenData** function

and in the **PENDATAHEADER** structure member **wPndts**. The scaling values do not behave in the same way as the Windows scaling units with similar names. For example, a 1-inch line in MM_HIENGLISH will not necessarily be an inch long on the screen, because GDI does not know the size of the monitor. However, with PDTS_HIENGLISH in **MetricScalePenData**, a line drawn an inch long is actually an inch long.

See Also

CreatePenData, **DestroyPenData**, PDTS_, PDK_

CreatePenDataHRC Overview

Overview

2.0

Returns the handle to the **HPENDATA** object containing the pen data in the **HRC**.

HPENDATA CreatePenDataHRC(HRC *hrc* **)**

Parameters

hrc

Handle to the **HRC** object.

Return Value

Returns a handle to the **HPENDATA** object if successful; otherwise, it returns NULL.

Comments

It is the responsibility of the caller to destroy the **HPENDATA** object.

A recognizer is not required to use or maintain OEM data; that is, a recognizer can choose to ignore some or all of the OEM data it receives from **AddPenDataHRC** or **AddPenInputHRC**. This means that the **HPENDATA** object the recognizer returns through **CreatePenDataHRC** may not contain all the OEM data originally provided by the application. Whether or not a recognizer uses the OEM data, it should store all such data it receives and forward it so that subsequent recognizers, if any, can use the data.

See Also

AddPenInputHRC, **AddPenDataHRC**

CreatePenDataRegion Overview

Overview

2.0

Creates a region that envelops the point data in an **HPENDATA** object.

HRGN CreatePenDataRegion(HPENDATA *hpndt*, **UINT** *uType* **)**

Parameters

hpndt

Handle to the **HPENDATA** object.

uType

Type of region to create. This can be one of the following values:

CPDR_BOX

The bounding box of the pen data ink is converted to a region.

CPDR_LASSO

The pen data describes a lasso that makes up the boundary of the region. If the last point of the pen data does not coincide with the first point, a closed figure is created either by joining the endpoints with a straight line or by using the intersection point of the beginning and ending line segments, whichever is more appropriate. Only the first stroke is used; if the pen data has more than a single stroke, subsequent strokes are ignored.

Return Value

This function returns a handle to a region if successful; otherwise the return value is NULL.

Comments

The coordinates of the region are the same as those used in the pen data. It is the application's responsibility to remove the region when the application is finished with it, using the Windows **DeleteObject** function.

CreatePenDataRegion enables an application to determine the screen area a gesture such as lasso or cut applies to. For an example of how to use the **CreatePenDataRegion** function to determine the area of a gesture, see the section "DoDefaultPenInput Messages" in Chapter 2, "Starting Out with System Defaults."

DestroyHRC Overview

Overview

2.0

Destroys an **HRC** object. A recognizer must export this function.

int DestroyHRC(HRC *hrc* **)**

Parameters

hrc

Handle to the **HRC** object.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.

Comments

If the **HRC** contains other objects such as an **HPENDATA** object, the recognizer must destroy the contained objects as well. After **DestroyHRC** returns HRCR_OK, the handle *hrc* is no longer valid. The application should set *hrc* to NULL to ensure it is not inadvertently used again.

See Also

CreateCompatibleHRC, **DestroyHRCRESULT**

DestroyHRCRESULT Overview

Overview

2.0

Destroys an **HRCRESULT** object. A recognizer must export this function.

int DestroyHRCRESULT(HRCRESULT *hrcresult* **)**

Parameters

hrcresult

Handle to the **HRCRESULT** object to destroy.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.

Comments

A recognizer must maintain a count of the number of **HRCRESULT** objects it creates. If an application calls **DestroyHRC**, the recognizer should not remove the **HRC** from memory until the application has called **DestroyHRCRESULT** for all **HRCRESULT** objects associated with the **HRC**.

After **DestroyHRCRESULT** returns HRCR_OK, the handle *hrcresult* is no longer valid. The application should set *hrcresult* to NULL to ensure it is not inadvertently used again.

See Also

GetResultsHRC, **DestroyHRC**

DestroyHWL Overview

Overview

2.0

Destroys a handle to a handwriting-recognition word list.

int DestroyHWL(HWL *hwl* **)**

Parameters

hwl

Word list to destroy.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

After **DestroyHWL** returns HRCR_OK, the handle *hwl* is no longer valid. The application should set *hwl* to NULL to ensure it is not inadvertently used again.

See Also

CreateHWL

DestroyInkset Overview

Overview

2.0

Frees memory associated with an inkset.

BOOL DestroyInkset(HINKSET *hinkset* **)**

Parameters

hinkset

Handle of an inkset to destroy.

Return Value

Returns TRUE if successful; otherwise FALSE.

Comments

Once memory is freed, the handle *hinkset* is invalid. The application should set the handle to NULL.

See Also

CreateInkset, **INTERVAL**

DestroyPenData Overview

Overview

1.0 2.0

Frees the memory associated with a specified pen data memory block.

BOOL DestroyPenData(HPENDATA *hpndt* **)**

Parameters

hpndt

Handle to a pen data memory block to destroy.

Return Value

Returns TRUE if the memory was successfully freed; otherwise, FALSE.

Comments

Once the memory block is destroyed, the **HPENDATA** handle is no longer valid. The application should set the handle to NULL.

See Also

CreatePenData, **CreatePenDataEx**

DictionarySearch Overview

Overview

1.0 2.0

Performs a dictionary search for a version 1.0 recognizer.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

BOOL DictionarySearch(LPRC *lprc*, **LPSYE** *lpsye*, **int** *cSye*, **LPSYV** *lpsyv*, **int** *csyvMax* **)**

Parameters

lprc

Address of an **RC** structure.

lpsye

Address of an array of **SYE** symbol elements that constitute the symbol graph.

cSye

Number of **SYE** structures in the array.

lpsyv

Output buffer of **SYV** types. This parameter contains the return results of the dictionary search. A SYV_NULL value is always appended at the end of this buffer. Therefore, this parameter must have enough space for *csyvMax* + 1 **SYV** symbol values.

csyvMax

Size of the output buffer.

Return Value

Returns TRUE if any enumeration is found in a dictionary. It returns FALSE if a NULL dictionary was requested or none of the enumerations was found in any dictionary.

Comments

The **DictionarySearch** function uses the symbol graph pointed to by *lpsye*, performs a dictionary search based on the options set in *lprc*, and returns the result as an array of **SYV** symbol values in the buffer pointed to by *lpsyv*. The function returns the number of **SYV** elements copied, limited by the maximum specified in the *csyvMax* parameter.

DictionarySearch first passes the symbol graph with DIRQ_SYMBOLGRAPH to all the dictionaries in the **rglpdf** array in the specified **RC** structure. If none succeeds, the function enumerates the symbol graph in *lpsye* and searches through all of the dictionary functions for a match. The calling application can get suggestions by setting the RCO_SUGGEST flag in the **IRcOptions** field in the **RC** structure. When this flag is set and no enumeration is found in any of the dictionaries in the **rglpdf** array, **DictionarySearch** tries to get a suggestion from the dictionaries on the path. **DictionarySearch** takes the first suggestion offered by any dictionary and returns that as the result of the search. If there are no suggestions, the function returns the best enumeration. The best enumeration is obtained using the **FirstSymbolFromGraph** function.

If the option RCO_NOSPACEBREAK is set in the **IRcOptions** field of the specified **RC** structure, **DictionarySearch** treats the entire *lpsye* array as a single symbol graph. If this flag is not set, the function

breaks down the input symbol graph into tokens delimited by white space, performs the search sequence on each of them, and assembles the result in the *lpsyv* array.

This function uses the **EnumSymbols** function for enumeration and the **wTryDictionary** member in the **RC** structure to specify the maximum number of enumerations to search through for each symbol graph token.

See Also

EnumSymbols, **FirstSymbolFromGraph**, **SYE**, SYV_, **RC**

DoDefaultPenInput Overview

Overview

2.0

Initiates default handling of pen input.

int DoDefaultPenInput(HWND *hwnd*, **UINT** *wEventRef* **)**

Parameters

hwnd

Handle to the window initiating the default processing.

wEventRef

An identifier of a pen event in the input stream, from which input is begun. This identifier is the value returned from the **GetMessageExtraInfo** function.

Return Value

Returns one of the following values:

Constant	Description
PCMR_OK	Pen collection was successfully started.
PCMR_ALREADYCOLLECTING	**StartPenInput** has already been called for this session.
PCMR_APPTERMINATED	The application aborted input.
PCMR_ERROR	Illegal parameter or unspecified error.
PCMR_INVALID_PACKETID	Invalid packet identifier.
PCMR_SELECT	Press-and-hold was detected. Collection is not started.
PCMR_TAP	A pen tap was detected. Collection is not started.

Comments

DoDefaultPenInput simplifies the pen input process by including the following capabilities in a single call:

- Starts pen input by calling **StartPenInput**
- Starts inking by calling **StartInking**
- Saves the screen background overwritten by the ink
- Collects the pen input data
- Stops inking by calling **StopInking**
- Stops pen input by calling **StopPenInput**
- Targets the pen input data to windows
- Recognizes results
- Sends the recognition results to the targets

The default processing proceeds in three phases: initialization, data gathering, and termination. A set of submessages corresponds to each of the three phases.

- During the initialization phase, the system sends the WM_PENEVENT sub-message PE_SETTARGETS and potentially several PE_GETPCMINFO and PE_GETINKINGINFO messages. After the target or the **DefWindowProc** function handles these messages and returns a value of PCMR_OK to indicate success, the data-gathering phase begins.
- During the data-gathering phase, the window specified by the *hwnd* parameter starts to receive the core pen-input submessages PE_PENDOWN, PE_PENUP, and PE_PENMOVE. The window should let these submessages fall through to **DefWindowProc**, which translates them into the higher-level messages PE_BEGINDATA and PE_MOREDATA. These are sent to one of the windows specified in the **htrgTarget** members of the **TARGET** structures if targeting is in progress; otherwise, the messages are sent to *hwnd*.
- The termination phase begins when the pen input terminates. The target window should let the core termination messages PE_TERMINATING and PE_TERMINATED fall through to **DefWindowProc**. The PE_ENDDATA, PE_RESULT, and PE_ENDINPUT submessages are sent by **DefWindowProc** while processing PE_TERMINATED.

A return value of LRET_ABORT to any of the WM_PENEVENT submessages aborts the entire process of default input.

See Also

WM_PENEVENT, **StartPenInput**, **StartInking**, **StopPenInput**, **StopInking**

DPtoTP Overview

Overview

1.0 2.0

Converts an array of points in display coordinates to tablet coordinates.

BOOL DPtoTP(LPPOINT *lppt*, **int** *cPnt* **)**

Parameters

lppt

Address of an array of **POINT** structures to convert to tablet coordinates. This parameter cannot be NULL.

cPnt

Number of **POINT** structures to convert.

Return Value

Returns TRUE if the conversion was successful; otherwise, returns FALSE.

Comments

Because of possible rounding errors, the **DPtoTP** and **TPtoDP** functions are not guaranteed to be perfect inverses of each other.

The calling application must avoid overflow by passing in points that are within the limits of the current physical display.

See Also

TPtoDP

DrawPenData Overview

Overview

1.0 2.0

Displays the pen data in an **HPENDATA** object as a trail of visible ink.

void DrawPenData(HDC *hdc*, **LPRECT** *lprect*, **HPENDATA** *hpndt* **)**

Parameters

hdc

Handle to a device context. This parameter can also be the handle of a metafile.

lprect

Bounding rectangle of ink, in client coordinates. Can be NULL.

hpndt

Handle to a pen data object.

Return Value

This function does not return a value. If *hpndt* is NULL, **DrawPenData** does nothing.

Comments

DrawPenData draws the pen data in the specified device context using the GDI **Polyline** function. The current settings in the device context rather than the ink characteristics determine how the data is rendered. This means the ink width and color specified in the **PENDATAHEADER** structure have no effect on how **DrawPenData** renders the ink. To alter the display characteristics of the ink, an application must call the appropriate Windows GDI functions to set the GDI drawing pen (not to be confused with the real pen).

The application using **DrawPenData** must either scale the data points or set the mapping appropriately if *lprect* is NULL.

If *lprect* is not NULL, the points are scaled into *lprect* as the drawing is done. Internally, nondestructive calls to the **SetViewportExtEx**, **SetViewPortOrg**, **SetWindowOrg**, and **SetWindowExtEx** functions are used to render the pen data in the device context within the bounds of the provided rectangle. An application must compute the proper pen width (if it is other than 1) before calling this function with a valid *lprect* parameter to account for the scaling that occurs.

DrawPenData draws the ink in the rectangle relative to the upper-left corner of the window. It ignores any changes that have been made to the origin of the device context by previous calls to the **SetWindowOrg** or **SetViewportOrgEx** functions. If the origin has changed, the rectangle passed to **DrawPenData** must be offset by the appropriate amount.

If the ink is to be drawn with a width of greater than 1 pixel, the width of the currently selected pen must be set to achieve the desired result. The width must be set in client coordinates if a mapping mode is set in the device context. For example, if the mapping mode has been set to MM_HIENGLISH, the pen width must be set to a number appropriate for the desired width in MM_HIENGLISH units to preserve the proper scale of the ink. This scaling is only an issue when the ink width is greater than 1.

The rendering of the ink data produced by **DrawPenData** generally does not exactly match the rendering produced by the display driver when the data was first collected. This discrepancy results because **DrawPenData** and the **Polyline** function use different algorithms to draw the data. The difference is an occasional "off by one" error that appears as a shifting of some pixels around the edges, depending on

the rounding done by **Polyline**. An application that requires an exact replication of the original ink rendering should call the **RedisplayPenData** function.

The **DrawPenDataEx** function allows more control when drawing the contents of pen structures.

See Also

CreatePenData, **DrawPenDataEx**, **DuplicatePenData**, **RedisplayPenData**

DrawPenDataEx Overview

Overview

2.0

An enhanced version of **DrawPenData**. Besides displaying the pen data in an **HPENDATA** object as a trail of visible ink, **DrawPenDataEx** can govern the speed at which the data is rendered, a process called *animation*.

int DrawPenDataEx(HDC *hdc*, **LPRECT** *lprectVP*, **HPENDATA** *hpndt*, **UINT** *iStrkFirst*, **UINT** *iStrkLast*, **UINT** *iPntFirst*, **UINT** *iPntLast*, **ANIMATEPROC** *lpfnAnimateCB*, **LPANIMATEINFO** *lpai*, **UINT** *fuFlags* **)**

Parameters

hdc

Handle to a device context.

lprectVP

Viewport rectangle, usually the bounding rectangle of the pen data, in client coordinates. The ink is scaled to fit the specified rectangle. If this parameter is NULL, the bounding rectangle of the ink in *hpndt* is used, in whatever coordinate system it happens to be in.

hpndt

Handle to a pen data object.

iStrkFirst

Index of the first stroke to display.

iStrkLast

Index of the last stroke to display.

iPntFirst

Index of the first point in the first stroke to display.

iPntLast

Index of the last point in the last stroke to display.

lpfnAnimateCB

Pointer to a callback function instance, or NULL. The callback function is called periodically during drawing, and animation is controlled by values in the structure addressed by the next parameter, *lpai*, which should not be NULL. If *lpfnAnimateCB* and the speed in the *lpai* structure parameters are NULL, the specified pen data is drawn without regard to timing information, and no callback functions are generated. See **AnimateProc** for a description of the callback function.

lpai

Address of an **ANIMATEINFO** structure that specifies animation parameters to control how the pen data is drawn. If this parameter is NULL, the function draws the specified pen data without regard to timing information, and no callback functions are generated; otherwise, the caller must initialize the **cbSize** member to sizeof(ANIMATEINFO).

fuFlags

This flag can be 0 or one of the following values:

DPD_HDCPEN

Use the GDI pen already selected into the specified device context. If this flag is set, any pen formatting stored in *hpndt* is ignored and all strokes are drawn with a single width and color. The **DrawPenData** function uses this flag.

DPD_DRAWSEL

Paint selected strokes in the specified range. A solid pen is used, with a width slightly larger than the stroke width. This flag can be used only for drawing and is ignored for animation. It is incompatible with DPD_HDCPEN.

Return Value

Returns PDR_OK if successful. Attempting to draw an empty **HPENDATA** (containing no strokes) also returns PDR_OK. Otherwise, returns one of the following:

Constant	Description
PDR_ABORT	Drawing aborted because pen data became invalid after a callback or yield.
PDR_CANCEL	Callback cancel or impasse. An impasse occurs when the user attempts to animate with 0 percent speed (that is, pause), but the callback interval is on a per-stroke basis.
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Bad animation structure, invalid sampling rate (0 or less) in pen data header, illegal flags, or other error.
PDR_MEMERR	Memory error.
PDR_PNDTERR	Invalid pen data. This value is also returned if the pen data is destroyed or corrupted during drawing or animation. This error can occur if an application is drawing a large pen data object and then destroys the data before drawing is complete.
PDR_PNTINDEXERR	Invalid point index.
PDR_STRKINDEXERR	Invalid stroke index.
PDR_VERSIONERR	Could not convert old pendata.

Comments

DrawPenDataEx is a general-purpose drawing function for rendering pen data objects. The calling application can use the timing information in the strokes to animate the pen data and specify which subset of the pen data should be drawn.

Partial pen data objects can be drawn by specifying first and last strokes and points with *iStrkFirst*, *iStrkLast*, *iPntFirst*, and *iPntLast*. Set beginning values to 0 and ending values to IX_END to display the entire pen data object. The function fails if any of these values lie outside the ranges available in the pen data. The stroke values must be between 0 and the total number of strokes in the pen data, and the point indices must be between 0 and the number of points in their stroke.

DrawPenDataEx can display only a set of sequential strokes with a single call. To draw nonsequential strokes—say, the second, fifth, and eighth strokes of the pen data—requires multiple calls to **DrawPenDataEx**.

Ink displayed by **DrawPenDataEx** differs slightly from the original rendering, as described in the **DrawPenData** topic. However, **DrawPenDataEx** can automatically display the ink with its original color and width, saving the application the burden of resetting the current GDI pen characteristics. To draw the

ink according to the GDI settings, set *fuFlags* to DPD_HDCPEN.

If *lpfnAnimateCB* is not NULL, the specified callback function must return TRUE to continue drawing, or FALSE to terminate drawing.

An application can modify the pen data while it is being rendered, for example, during an animation callback, task switching, or internal yield. However, doing so can make internal pointers or data invalid and result in unpredictable behavior. For this reason, editing the pen data during rendering is not recommended.

See Also

AnimateProc, **DrawPenData**, **RedisplayPenData**, **DrawPenDataFmt**, **ANIMATEINFO**

DrawPenDataFmt Overview

Overview

2.0

The **DrawPenDataFmt** macro is used to draw pen data using its stored stroke attributes.

int DrawPenDataFmt(HDC *hdc*, **LPRECT** *lprectVP*, **HPENDATA** *hpndt* **)**

Parameters

hdc

Handle to a device context.

lprectVP

Viewport rectangle, usually the bounding rectangle of the **HPENDATA** object, in client coordinates. The ink is scaled to fit the specified rectangle. If this parameter is NULL, the bounding rectangle of the ink in *hpndt* is used, in whatever coordinate system it happens to be in.

hpndt

Handle to an **HPENDATA** object.

Return Value

Returns PDR_OK if successful. Attempting to draw valid but empty pen data (containing no strokes) also returns PDR_OK. Otherwise, the return value is one of the following:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Invalid sampling rate (0 or less) in pen data header, or other error.
PDR_MEMERR	Memory error.
PDR_PNDTERR	Invalid pen data.
PDR_VERSIONERR	Could not convert old pen data.

Comments

The **DrawPenDataFmt** macro is a wrapper for **DrawPenDataEx**, providing default values for most of the parameters.

The definition is:

```
#define DrawPenDataFmt( hdc, lprectVP, hpndt )
DrawPenDataEx( hdc, lprectVP, hpndt, 0, IX_END, 0, IX_END, NULL,
                    NULL, 0 );
```

These default values specify:

- Full-speed rendering (no animation).
- Entire data set is drawn (no stroke subsets).

See Also

DrawPenDataEx

DuplicatePenData Overview

Overview

1.0 2.0

Duplicates an **HPENDATA** object, allowing an application to generate clones of existing pen data.

HPENDATA DuplicatePenData(HPENDATA *hpendata*, **UINT** *gmemFlags* **)**

Parameters

hpendata

Pen data to be duplicated.

gmemFlags

Flag that specifies whether or not the Windows **GlobalAlloc** function should create a shared memory object when the pen data object is created. This should be either 0 or GMEM_DDESHARE. The GMEM_MOVEABLE and GMEM_ZEROINIT flags are added to this value and other GMEM_ flags are ignored.

Return Value

Returns a handle to the duplicated pen data object if successful; otherwise, it returns NULL. It returns NULL if memory is not allocated successfully.

Comments

The **DuplicatePenData** function duplicates the data specified by the *hpendata* parameter by creating a second pen data memory block. The application is responsible for destroying this memory block by calling **DestroyPenData**.

See Also

CreatePenData, **DestroyPenData**

EmulatePen Overview

Overview

1.0 2.0

Emulates a pen in an application that does not use the standard Windows I-beam cursor in text areas.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions. Use **DoDefaultPenInput** or hedit controls instead.

void EmulatePen(BOOL *fPen* **)**

Parameters

fPen

Flag to set pen emulation. TRUE activates pen emulation; FALSE turns it off.

Return Value

This function does not return a value.

Comments

The application must call **EmulatePen** with *fPen* set to TRUE whenever the cursor is over a text input window. When the cursor leaves that area, the application must call **EmulatePen** with *fPen* set to FALSE.

EmulatePen is useful only for those applications that do not use other Pen API services and do not use the standard Windows I-beam cursor. Windows automatically provides pen-based input in edit controls that use the I-beam cursor, as described in Chapter 1.

See Also

DoDefaultPenInput

EnableGestureSetHRC Overview

Overview

2.0

Enables or disables recognition of specific gestures or collections of gestures in an **HRC** object.

int EnableGestureSetHRC(HRC *hrc*, **SYV** *syv*, **BOOL** *fEnable* **)**

Parameters

hrc

Handle to the **HRC** object.

syv

Either a gesture SYV_ symbol value, such as SYV_COPY, or one or more of the following GST_ constants combined using the bitwise-OR operator. Note that individual SYV_ gesture symbol values cannot be combined with GST_ constants.

Constant	Description
GST_SEL	Selection and lasso.
GST_CLIP	Cut, copy, paste.
GST_WHITE	Space, tab, return.
GST_EDIT	Insert, correct, undo.
GST_CIRCLELO	Lowercase circle gestures.
GST_CIRCLEUP	Uppercase circle gestures.
GST_CIRCLE	All circle gestures.
GST_ALL	All gestures.

fEnable

Enable recognition flag. This flag must be set to TRUE to enable recognition of the gesture or gestures in *syv*, or to FALSE to disable recognition of the specified gestures.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

The results of **EnableGestureSetHRC** are cumulative. The function can be called several times in succession to refine the precise gesture set required. However, calling **EnableGestureSetHRC** with *syv* set to GST_ALL and *fEnable* set to FALSE disables all gestures.

By default, a recognition context **HRC** enables all gestures that its associated recognizer supports.

Example

The following example enables selection, Clipboard functions, and SYV_CIRCLEUPA:

```
EnableGestureSetHRC( hrc, GST_ALL, FALSE );        // Disable all
EnableGestureSetHRC( hrc, GST_SEL | GST_CLIP, TRUE ); // Enable sets
EnableGestureSetHRC( hrc, SYV_CIRCLEUPA, TRUE );         // Enable circle A
```

See Also

SetAlphabetHRC, SYV_

EnableSystemDictionaryHRC Overview

Overview

2.0

Enables or disables a recognizer's dictionary.

int EnableSystemDictionaryHRC(HRC *hrc*, **BOOL** *fEnable* **)**

Parameters

hrc

Handle to the **HRC** object for the recognizer.

fEnable

Enable recognition flag. This flag must be set to TRUE to enable use of the dictionary, or FALSE to disable its use.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter, no system dictionary, or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

The enable state of the system dictionary does not affect any word lists that may be set into the **HRC** object.

See Also

SetWordlistHRC

EndEnumStrokes Overview

Overview

1.0 2.0

Unlocks an **HPENDATA** memory block previously locked with the function **BeginEnumStrokes**.

Note This function is provided only for compatibility with version 1.0 of the Pen API, and will not be supported in future versions.

LPPENDATA EndEnumStrokes(HPENDATA *hpndt* **)**

Parameters

hpndt

Handle to the locked **HPENDATA** memory block.

Return Value

Returns NULL if the function is successful; otherwise, the return value is nonzero.

Comments

EndEnumStrokes internally calls the Windows **GlobalUnlock** function to unlock the memory block specified by *hpndt*. Calling **EndEnumStrokes** invalidates any pointers previously returned by the **GetPenDataStroke** function.

See Also

BeginEnumStrokes, **GetPenDataStroke**

EndPenInputHRC Overview

Overview

2.0

Informs a recognizer that pen data input has been terminated. A recognizer must export this function.

int EndPenInputHRC(HRC *hrc* **)**

Parameters

hrc

Handle to the **HRC** object for the recognizer.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_MEMERR	Insufficient memory.
HRCR_ERROR	Invalid parameter or other error.

Comments

EndPenInputHRC does not instruct the recognizer to complete recognition; an application must call **ProcessHRC** to do that. However, an application that does not use **DoDefaultPenInput** must call **EndPenInputHRC** when it detects that input has finished. (**DoDefaultPenInput** calls **EndPenInputHRC** internally.)

The recognizer can terminate open-ended states and reduce ambiguity in searches when it knows that no more ink will arrive. For example, the recognizer can keep various options open for possible delayed strokes that can modify a character. **EndPenInputHRC** tells the recognizer that no more delayed strokes will arrive.

After calling **EndPenInputHRC** for an **HRC**, an application should cease adding pen input into the **HRC**. Some recognizers, such as the Microsoft Handwriting Recognizer (GRECO.DLL), do not accept late pen input. If the application calls **AddPenInputHRC** after having called **EndPenInputHRC** for the same **HRC**, the Microsoft Handwriting Recognizer returns HRCR_ERROR.

Other recognizers may differ. With such recognizers, a client may continue to add pen input without error into a recognition context, even after having called **EndPenInputHRC**. However, doing so is not efficient. In the worst case, the recognizer may be forced to reprocess all of the pen data from the beginning.

For an example of a normal termination sequence, see the code sample in **GetSymbolsHRCRESULT**.

See Also

ProcessHRC, **DoDefaultPenInput**

EnumSymbols Overview

Overview

1.0 2.0

Enumerates strings in a symbol graph in order of most probable to least probable.

UINT EnumSymbols(LPSYG *lpsyg*, **UINT** *cstrMax*, **ENUMPROC** *lpEnumFunc*, **LPVOID** *lvData* **)**

Parameters

lpsyg

Address of the symbol graph **SYG**.

cstrMax

Maximum number of strings to enumerate.

lpEnumFunc

Address of enumeration function.

lvData

Application-specific data.

Return Value

Returns the number of strings enumerated.

Comments

The **EnumSymbols** function enumerates all symbol strings (to a maximum defined by *cstrMax*) contained in the symbol graph that *lpsyg* points to. The *lpEnumFunc* parameter points to the enumeration function called with each enumeration.

To generate all the symbols from a symbol graph, set *cstrMax* equal to the value retrieved by passing *lpsyg* to **GetSymbolCount**.

See Also

EnumSymbolsCallback, **FirstSymbolFromGraph**, **SYG**, SYV_

EnumSymbolsCallback

1.0 2.0

EnumSymbolsCallback is a callback function pointed to by the *lpEnumFunc* parameter of **EnumSymbols**. The callback function can have any name. The function's name must appear in the EXPORT section of the application's module definition file.

int CALLBACK EnumSymbolsCallback(LPSYV *lpsyv*, **int** *csyv*, **FAR void *** *lvData* **)**

Parameters

lpsyv

Symbol string.

csyv

Count of symbols in string.

lvData

Address of application-specific data from **EnumSymbols**.

Return Value

Returns TRUE to continue enumeration, or FALSE to stop enumeration.

See Also

EnumSymbols, SYV_

ExtractPenDataPoints Overview

Overview

2.0

Extracts points from a specified stroke in an **HPENDATA** object.

int ExtractPenDataPoints(HPENDATA *hpndt*, **UINT** *iStrk*, **UINT** *iPnt*, **UINT** *cPnts*, **LPPOINT** *lppt*, **LPVOID** *lpvOem*, **UINT** *fuOption* **)**

Parameters

hpndt

Handle to an **HPENDATA** object.

iStrk

Zero-based index of the stroke to remove points from.

iPnt

Zero-based index to the first point to remove.

cPnts

Count of points to remove. If this value is greater than the number of points after *iPnt*, all the points from *iPnt* to the last point of the stroke are removed. **ExtractPenDataPoints** fails if *iPnt* is greater then the number of points in the stroke.

lppt

Array of **POINT** structures that receives the extracted points. This must be large enough to hold *cPnts* points.

lpvOem

Buffer to put extracted OEM data if it exists, or NULL. This must be large enough to hold *cPnts* OEM packets.

fuOption

Flags. This value can be EPDP_REMOVE to remove the points from the stroke in the pen data object.

Return Value

Returns PDR_OK if successful; otherwise, the return value can be one of the following negative values:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Out of memory.
PDR_STRKINDEXERR	Invalid stroke index.
PDR_PNTINDEXERR	Invalid point index.
PDR_VERSIONERR	Could not convert old pen data object.

Comments

ExtractPenDataPoints extracts points (and OEM data, if any) from a specified stroke of the pen data object specified by *hpndt*. It copies the extracted points and the OEM data to the buffers pointed to by *lppt*

and *lpvOem*.

Use **ExtractPenDataStrokes** to extract strokes from the pen data object or **RemovePenDataStrokes** to remove strokes from the pen data object.

See Also

InsertPenDataPoints, **InsertPenDataStroke**, **RemovePenDataStrokes**

ExtractPenDataStrokes Overview

Overview

2.0

Creates a new **HPENDATA** object that is a subset of an existing object.

int ExtractPenDataStrokes(HPENDATA *hpndt*, **UINT** *fuExtract*, **LPARAM** *lParam*, **LPHPENDATA** *lphpndtNew*, **UINT** *gmemFlags* **)**

Parameters

hpndt

Handle to an existing pen data object.

fuExtract

Extraction options and modifiers. This value can be a combination of one of the principal EPDS_ options, with an optional comparison modifier, if appropriate, and the optional EPDS_REMOVE modifier. The flags should be combined using the bitwise-OR operator.

The following table gives the principal options. These options specify what the new pen data object will be based on.

Constant	Description
EPDS_INKSET	Based on a handle to an inkset. The EPDS_GT, EPDS_GTE, EPDS_LT, and EPDS_LTE comparison operators are ignored, because extraction is based on matching the inkset. (However, EPDS_NOT creates a pen data set with all stroke/inkset intersections that do not match the provided inkset.)
EPDS_PENTIP	Based on complete pen-tip characteristics.
EPDS_SELECT	Based on selected strokes.
EPDS_STROKEINDEX	Based on index.
EPDS_TIPCOLOR	Based on pen-tip color.
EPDS_TIPWIDTH	Based on pen-tip width.
EPDS_TIPNIB	Based on pen tip nib style.
EPDS_USER	Based on user-specific value.
	The following table gives the optional comparison modifiers and the optional removal modifier:
EPDS_LT	Less than comparison: extract all strokes with attribute less than the value specified in *lParam*.
EPDS_LTE	Less than or equal comparison: extract all strokes with attributes less than or equal to the value specified in *lParam*.
EPDS_GT	Greater than comparison: extract

	all strokes with attributes greater than the value specified in *lParam*.
EPDS_GTE	Greater than or equal comparison: extract all strokes with attributes greater than or equal to the value specified in *lParam*.
Constant	**Description**
EPDS_NOT	Negative comparison (alias EPDS_NE): extract all strokes with attributes not equal to the value specified by *lParam*. If combined with other EPDS_ constants, reverses the constant meaning (for example, EPDS_NE | EPDS_LT | EPDS_GT means not less than or not greater than). If *lParam* is EPDS_SELECT, EPDS_NOT means extract all unselected strokes.
EPDS_REMOVE	Remove matching strokes from source. If this flag is added, any strokes matching the criteria for extraction are removed from the source pen data.

lParam

Meaning is dependent on the value of the *fuExtract* parameter, as follows:

Constant	Description
EPDS_INKSET	*lParam* is a handle to an inkset.
EPDS_PENTIP	*lParam* is a pointer to a **PENTIP** structure to compare. Only equal or not-equal matches are supported. EPDS_LT, for example, is ignored.
EPDS_SELECT	*lParam* is not used and should be set to 0.
EPDS_STROKEINDEX	*lParam* is a zero-based stroke index to compare.
EPDS_TIPCOLOR	*lParam* is a pen tip color to compare.
EPDS_TIPNIB	*lParam* is a pen tip nib style to compare. Only equal or not-equal matches are supported.
EPDS_TIPWIDTH	*lParam* is a pen tip width to compare.
EPDS_USER	*lParam* is a user-specific value to compare, cast to a double-word value.

lphpndtNew

Address of a pen data handle if one is to be created; otherwise, NULL.

gmemFlags

Flag that specifies whether or not the Windows **GlobalAlloc** function should create a shared memory object when the pen data object is created. This should be either 0 or GMEM_DDESHARE. The GMEM_MOVEABLE and GMEM_ZEROINIT flags are added to this value, and other GMEM_ flags are ignored.

Return Value

Returns the number of strokes that match the comparison criteria if successful, or a negative error value. (The return value can be 0. The maximum is the largest integer value.) The error value can be one of the following:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_INKSETERR	Invalid inkset and EPDS_INKSET specified.
PDR_MEMERR	Memory error.
PDR_NA	Option not available.
PDR_PNDTERR	Invalid pendata.
PDR_STRKINDEXERR	Invalid stroke index.
PDR_USERDATAERR	EPDS_USER was specified but there is no per-stroke user data.
PDR_VERSIONERR	Could not convert old pendata.

Comments

ExtractPenDataStrokes extracts strokes from an existing pen data object, optionally creating a new pen data object made up of the extracted strokes. The extraction can be a copy or move process; that is, the source pen data object can remain the same or contain only the remaining strokes not moved to the new structure. Modifier flags in *fuExtract* specify how the value in *lParam* compares with attributes of the pen data strokes (equal by default, greater than, less than, or none of these three).

If *lphpndtNew* is NULL, no pen data object is created. This is useful for modifying the original pen data object *hpndt* (when EPDS_REMOVE specified), or simply for determining a return value without modifying or creating a pen data object. If *lphpndtNew* is not NULL, the flags specified by *gmemFlags* are passed to the **GlobalAlloc** function when memory for the pen data memory block is created.

If EPDS_REMOVE is specified, any strokes with an attribute matching the comparison criteria are removed from the source pen data object, regardless of whether a new pen data is created. In the case of inksets, this may actually generate more strokes if there are multiple intersections with the inkset within any one stroke.

Example

To create an **HPENDATA** object consisting only of selected strokes:

```
ExtractPenDataStrokes( hpndt, EPDS_SELECT, 0, &hpndtDst, 0 );
```

To return the count of selected strokes:

```
ExtractPenDataStrokes( hpndt, EPDS_SELECT, 0, NULL, 0 );
```

To delete all but the selected strokes from the source:

```
ExtractPenDataStrokes( hpndt, EPDS_NOT | EPDS_SELECT | EPDS_REMOVE,
                               0, NULL, 0 );
```

To copy strokes 0 through 10 inclusive to a new **HPENDATA** object:

```
ExtractPenDataStrokes( hpndt, EPDS_LTE | EPDS_STROKE, 10,
                                &hpndtDst, 0 );
```

To move all but blue strokes to a separate **HPENDATA** object:

```
ExtractPenDataStrokes( hpndt, EPDS_NOT | EPDS_TIPCOLOR | EPDS_REMOVE,
                                RGB_BLUE, &hpndtDst, 0 );
```

See Also

DuplicatePenData

FirstSymbolFromGraph Overview

Overview

1.0 2.0

Retrieves an array of symbols that is the most likely interpretation of a specified symbol graph **SYG**.

void FirstSymbolFromGraph(LPSYG *lpsyg*, **LPSYV** *lpsyv*, **int** *cSyvMax*, **LPINT** *lpcSyv* **)**

Parameters

lpsyg

Address of the symbol graph.

lpsyv

Address of an empty array of SYV_ symbol values. **FirstSymbolFromGraph** fills this array with the likeliest interpretation from the graph.

cSyvMax

Size of the array that *lpsyv* points to.

lpcSyv

Number of symbols returned in *lpsyv*. This value is 0 if *lpsyg* is empty. It is -1 if the buffer is not large enough to hold the results.

Return Value

This function does not return a value.

Comments

The array of symbols is identical to the first string returned to the **EnumSymbolsCallback** callback function of **EnumSymbols**.

See Also

EnumSymbols, **SYG**, SYV_

GetAlphabetHRC Overview

Overview

2.0

Retrieves the alphabet being used in a handwriting recognition context **HRC**.

int GetAlphabetHRC(HRC *hrc*, **LPALC** *lpalc*, **LPBYTE** *rgbfAlc* **)**

Parameters

hrc

Handle to the **HRC** object.

lpalc

Address of a buffer that receives the current ALC_ values. If NULL, this parameter is ignored.

rgbfAlc

Address of an array of bits or NULL. If NULL, this parameter is ignored. If *lpalc* contains ALC_USEBITMAP and *rgbfAlc* points to a valid array, the array is filled according to the bits set by the **SetAlphabetHRC** function.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

If *rgbfAlc* is not NULL, the array it points to must be large enough to accommodate 256 bits (32 bytes). If the *n*th bit is set, the *n*th ANSI character is recognizable. Bits representing characters with ASCII values less than 32 (the space character) currently have no meaning.

ALC_DEFAULT specifies the set of characters at or above ALC_SYSMINIMUM that the recognizer can accurately distinguish.

For a description of alphabets and their relationship to a recognizer, see "Configuring the HRC" in Chapter 5, "The Recognition Process."

See Also

EnableGestureSetHRC, **SetAlphabetHRC**, **GetAlphabetPriorityHRC**, ALC_

GetAlphabetPriorityHRC Overview

Overview

2.0

Retrieves the alphabet priority used in a handwriting recognition context **HRC**.

int GetAlphabetPriorityHRC(HRC *hrc*, **LPALC** *lpalc*, **LPBYTE** *rgbfalc* **)**

Parameters

hrc

Handle to the **HRC** object.

lpalc

Address of an **ALC** type that will be filled with the current ALC_ priority values.

rgbfalc

Address of a 256-bit (32-byte) buffer whose bits map to ANSI single-byte characters, or NULL if this information is not required.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

For a description of how a recognizer uses alphabet priority, see "Configuring the HRC" in Chapter 5, "The Recognition Process."

See Also

GetAlphabetHRC, **SetAlphabetPriorityHRC**, ALC_

GetAlternateWordsHRCRESULT Overview

Overview

2.0

Returns alternative word interpretations of a previous result. (Not supported in Japanese version.)

int GetAlternateWordsHRCRESULT(HRCRESULT *hrcresult***, UINT** *iSyv***, UINT** *cSyv***, LPHRCRESULT** *rghrcresults***, UINT** *cResults* **)**

Parameters

hrcresult

Handle of a results object.

iSyv

Index of the first of a span of symbols within the results object.

cSyv

The number of symbols in the original result, starting at *iSyv*, for which alternative words are required.

rghrcresults

Address of a result array. This address cannot be NULL.

cResults

The size of the *rghrcresults* array in results. This parameter must be greater than 0.

Return Value

Returns the number of results actually provided, if successful. This can be less than the space allocated in *rghrcresults*, and may be 0; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

GetAlternateWordsHRCRESULT provides alternative word interpretations of a previous result. The alternatives returned are strongly coerced to words in the recognizer's dictionary, if enabled, and the word list, if any, of the **HRC** that processed the results.

The span of symbols defined by *iSyv* and *cSyv* need not fall on word boundaries. However, the recognizer returns only a single word per result. It is the application's responsibility to ensure that embedding a full word within other symbols makes sense. (The application can also choose to let the user make that decision.) For example, finding alternatives for "polce" in the phrase "pig-in-a-polce" could legitimately return "poke" as an alternative, but alternatives for "kef" in "markefplace" would probably be meaningless.

See Also

GetResultsHRC

GetBoxMappingHRCRESULT Overview

Overview

2.0

Returns the box indices for a range of symbols.

int GetBoxMappingHRCRESULT(HRCRESULT *hrcresult***, UINT** *iSyv***, UINT** *cSyv***, UINT FAR *** *rgi* **)**

Parameters

hrcresult

Handle of a results object.

iSyv

Index of the first symbol of interest in the results object.

cSyv

The number of symbols following *iSyv* for which box indices are required. Note that the array *rgi* must be large enough to accommodate this many items of size UINT. A value of 0 is allowed, in which case the function simply returns 0.

rgi

Address of an index array. The array must be large enough to store *cSyv* indices. This address cannot be NULL.

Return Value

Returns the number of indices actually retrieved, if successful. This can be less than the space allocated in *rgi* if *iSyv* indexes an element near the end of the results array, and is 0 if *iSyv* indexes a nonexistent element; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

GetBoxMappingHRCRESULT is typically used with boxed input established by **SetGuideHRC**. If no guide structure has been set, the recognizer will return HRCR_ERROR.

It is possible to allocate a small buffer in *rgi* and call **GetBoxMappingHRCRESULT** repeatedly, incrementing the index *iSyv* each time by the number of indices returned in the previous call until **GetBoxMappingHRCRESULT** returns 0.

See Also

GetResultsHRC, **SetGuideHRC**

GetBoxResultsHRC Overview

Overview

2.0

Encapsulates recognizer functionality for boxed input.

int GetBoxResultsHRC(HRC *hrc*, **UINT** *cAlt*, **UINT** *iSyv*, **UINT** *cBoxRes*, **LPBOXRESULTS** *rgBoxResults*, **BOOL** *fGetInkset* **)**

Parameters

hrc

Handle to the **HRC** object used for the boxed input.

cAlt

Count of alternatives expected in the **BOXRESULTS** structure. If this parameter is 0, the function returns 0.

iSyv

Index to the starting symbol.

cBoxRes

The count of **BOXRESULTS** structures that the *rgBoxResults* array can hold. This parameter must be greater than 0.

rgBoxResults

Address of an array of **BOXRESULTS** structures.

fGetInkset

Flag to request inksets for each result if TRUE. If FALSE, the recognizer provides no inksets.

Return Value

Returns the count of **BOXRESULTS** elements returned in the *rgBoxResults* structure, if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_INVALIDGUIDE	The guide structure is invalid.
HRCR_MEMERR	Insufficient memory.
HRCR_HOOKED	A hook preempted the result.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

GetBoxResultsHRC simplifies the task of boxed recognition by providing character alternatives on a per-box basis in one call.

If *fGetInkset* is TRUE, the recognizer assigns a valid inkset handle to the **hinksetBox** member of the **BOXRESULTS** structure addressed by *rgBoxResults*. It is the application's responsibility to destroy these inksets with **DestroyInkset**.

Example

The following code sample gets results for 10 boxes at a time, with five alternatives per box:

```
HANDLE hMem = GlobalAlloc( GHND, 10 * (sizeof( BOXRESULTS )
            + (5-1) * sizeof( SYV )) );
LPBOXRESULTS rgBoxR = (LPBOXRESULTS)GlobalLock( hMem );
UINT indx = 0;

do
{
   int cRes = GetBoxResultsHRC( hrc, 5, indx, 10, rgBoxR, FALSE );
         .
         .        // Check for errors and use rgBoxR
         .
   indx += (UINT)cRes;
}
while (cRes == 10);
```

See Also

GetBoxMappingHRCRESULT, **GetResultsHRC**, **DestroyInkset**

GetGlobalRC Overview

Overview

1.0 2.0

Queries the current default settings and fills an **RC** structure with the global values. In version 2.0 of the Pen API, **RC** is made obsolete by the **HRC** object.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

UINT GetGlobalRC(LPRC *lprc*, **LPSTR** *lpszDefRecog*, **LPSTR** *lpszDefDict*, **int** *cbDefDictMax* **)**

Parameters

lprc

Address of an **RC** structure. This parameter can be NULL.

lpszDefRecog

Address of a character string in which the default recognizer module name is returned. This must be at least 128 bytes long. This parameter can be NULL.

lpszDefDict

Buffer in which the default dictionary path is returned. This path ends with two null characters. This parameter can be NULL.

cbDefDictMax

Size of *lpszDefDict* buffer to be filled.

Return Value

Returns GGRC_OK if successful; otherwise, the return value may be one of the following values:

Constant	Description
GGRC_PARAMERROR	One or more invalid parameters were detected. The call to **GetGlobalRC** has no effect.
GGRC_DICTBUFTOOSMALL	The size of the *lpszDefDict* buffer is not large enough to contain the entire dictionary path. The buffer is filled with as many complete dictionary module names as allowed by the size. The list is terminated by a null string.

Comments

GetGlobalRC fills the **RC** structure with global values. Values that have no default settings—for example, the bounding rectangle—are set to 0.

An application does not need to call this function to use the default values. When an application initializes an **RC** structure using **InitRC**, the system default values are set as the values for the structure members. This function returns the actual current values for **RC** members. The **InitRC** function returns the default values, which include placeholder values for some **RC** members.

See Also

InitRC, **SetGlobalRC**, **RC**

GetGuideHRC Overview

Overview

2.0

Retrieves the guide structure being used in a recognition context **HRC**.

int GetGuideHRC(HRC *hrc*, **LPGUIDE** *lpguide*, **UINT FAR *** *lpnFirstVisible* **)**

Parameters

hrc

Handle to the **HRC** object.

lpguide

Address of a **GUIDE** structure; all coordinates are in screen coordinates. This parameter cannot be NULL.

lpnFirstVisible

Pointer to first visible character or line, or NULL. For boxed controls, this is the first visible box (leftmost and topmost for left-right, top-down languages like English). For other controls, this is the first visible character position (left-most for English) in a single-line control, and the first visible line (topmost for English) in multiline controls.

If set to NULL, *lpnFirstVisible* is ignored.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

See Also

GUIDE, **SetGuideHRC**

GetHotspotsHRCRESULT Overview

Overview

2.0

Retrieves the hot spots for a particular symbol.

int GetHotspotsHRCRESULT(HRCRESULT *hrcresult***, UINT** *iSyv***, LPPOINT** *lppt***, UINT** *cPnts* **)**

Parameters

hrcresult

Handle of a results object.

iSyv

Index of the symbol in the results object.

lppt

Address of an array of up to MAXHOTSPOT **POINT** structures.

cPnts

Actual size of *lppt* array in points.

Return Value

If successful, returns the count of hot spots; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

Any symbol can have hot spots, but they are usually of interest only for gestures. For example, if the user writes "X" for deletion, the center of the "X"—its hotspot —points to the item to be deleted. Hot spots are returned in tablet coordinates. The maximum number of hot spots allowed is provided in the PENWIN.H constant MAXHOTSPOT. The Microsoft Handwriting Recognizer (GRECO.DLL), supports this function for gesture symbols only.

If *cPnts* is smaller than the actual number of hot spots, only *cPnts* points are reported.

GetHRECFromHRC Overview

Overview

2.0

Retrieves a handle to the recognizer bound to an **HRC** object. A recognizer must export this function.

HREC GetHRECFromHRC(HRC *hrc* **)**

Parameters

hrc

Handle to the **HRC** object.

Return Value

If successful, returns the handle to the recognizer used for the **HRC** object; otherwise, returns NULL.

See Also

CreateCompatibleHRC, **InstallRecognizer**

GetInksetInterval Overview

Overview

2.0

Retrieves an interval from an inkset.

int GetInksetInterval(HINKSET *hinkset*, **UINT** *uIndex*, **LPINTERVAL** *lpi* **)**

Parameters

hinkset

Handle to an inkset.

uIndex

Zero-based index of an interval or IX_END.

lpi

Pointer to an **INTERVAL** structure. This can be NULL if the user merely wishes to find out how many intervals are in the inkset.

Return Value

GetInksetInterval returns the number of intervals in the inkset if successful; otherwise, the return value can be one of the following negative values:

Constant	Description
ISR_ERROR	The inkset handle is bad, or a parameter error.
ISR_BADINDEX	The interval index is bad.
ISR_BADINKSET	The inkset has been corrupted or contains bad intervals.

Comments

An application can use **GetInksetInterval** to enumerate all the intervals in an inkset.

See Also

GetInksetIntervalCount, **INTERVAL**

GetInksetIntervalCount Overview

Overview

2.0

Returns the number of intervals in an inkset.

int GetInksetIntervalCount(HINKSET *hinkset* **)**

Parameters

hinkset

Handle to an inkset.

Return Value

Returns the number of intervals in the inkset is successful; otherwise, the return value can be one of the following negative values:

Constant	Description
ISR_ERROR	The inkset handle is bad, or a parameter error.
ISR_BADINKSET	The inkset has been corrupted or contains bad intervals.

Comments

An application uses **GetInksetIntervalCount** to determine how many intervals there are to enumerate in an inkset. This function can also be used to verify that an inkset is valid.

See Also

GetInksetInterval

GetInternationalHRC

2.0

Retrieves the country, language, script direction, and international preferences from a recognition context **HRC** object.

int GetInternationalHRC(HRC *hrc*, **UINT FAR *** *lpuCountry*, **LPSTR** *lpszLangCode*, **UINT FAR *** *lpfuFlags*, **UINT FAR *** *lpuDir* **)**

Parameters

hrc

Handle to the **HRC** object.

lpuCountry

The country code, or NULL to ignore this value.

lpszLangCode

A buffer large enough to receive a three-letter string (that is, 4 bytes) identifying the language ("enu", "fra", etc.). If set to NULL, *lpszLangCode* is ignored.

lpfuFlags

A pointer to a flags value or NULL to ignore this value. If **GetInternationalHRC** returns SIH_ALLANSICHAR in *lpfuFlags*, it means that the user intends to use the entire ANSI character set. If this is the case, the application should ignore the value returned in *lpszLangCode*, since all the ANSI-based languages are undifferentiated.

lpuDir

Address of a value for the script direction, or NULL to ignore this value. This value specifies which primary and secondary writing directions are in use. Possible values are:

Constant	Description
SSH_RD	To right and down (English).
SSH_RU	To right and up.
SSH_LD	To left and down (Hebrew).
SSH_LU	To left and up.
SSH_DL	Down and to the left (Chinese).
SSH_DR	Down and to the right (Chinese).
SSH_UL	Up and to the left.
SSH_UR	Up and to the right.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

See Also

SetInternationalHRC

GetMaxResultsHRC Overview

Overview

2.0

Gets the maximum number of recognition results that a recognizer can generate in the current handwriting recognition context **HRC** object.

int GetMaxResultsHRC(HRC *hrc* **)**

Parameters

hrc

Handle to the **HRC** object.

Return Value

If successful, returns the maximum number of recognition results as a positive number; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.

Comments

The default maximum number of results a recognizer can return is 1. An application must call **SetMaxResultsHRC** to set a different maximum value.

See Also

SetMaxResultsHRC

GetPenAppFlags Overview

Overview

2.0

GetPenAppFlags returns task pen flags cached by **RegisterPenApp**.

UINT GetPenAppFlags( )

Return Value

GetPenAppFlags returns the flags set by **RegisterPenApp** for the current task. It extends and replaces the functionality of the version 1.0 function **IsPenAware**, which will not be supported in future versions of the Pen API.

Applications written specifically for Windows 95 and later versions automatically get RPA_DEFAULT so that any edit controls created by such applications automatically become pen-aware.

If the registration cache has been destroyed (which indicates PENWIN.DLL has been unloaded), this function returns 0.

See Also

RegisterPenApp, **IsPenAware**

GetPenAsyncState Overview

Overview

1.0 2.0

Gets the state of the pen barrel button.

BOOL GetPenAsyncState(UINT *wPDK* **)**

Parameters

wPDK

One of the PDK_ values for the barrel buttons. The following table lists the PDK_ values that **GetPenAsyncState** can query for:

Constant	Description
PDK_BARREL1	Get state of barrel button 1.
PDK_BARREL2	Get state of barrel button 2.
PDK_BARREL3	Get state of barrel button 3.

Return Value

Returns TRUE if the specified barrel button state is currently down; otherwise, the return value is FALSE.

GetPenDataAttributes Overview

Overview

2.0

Retrieves information about an **HPENDATA** object.

int GetPenDataAttributes(HPENDATA *hpndt*, **LPVOID** *lpvBuffer*, **UINT** *uOption* **)**

Parameters

hpndt

Handle to the **HPENDATA** object.

lpvBuffer

Pointer to a structure whose type depends on *uOption*, or NULL if the *uOption* parameter does not require this buffer.

uOption

Specifies the attributes to retrieve. This parameter can be one of the following:

Constant	Description
GPA_MAXLEN	Retrieves the length (in points) of the longest stroke. *lpvBuffer* is unused and ignored.
GPA_POINTS	Retrieves the total number of points. *lpvBuffer* is unused and ignored.
GPA_PDTS	Retrieves the **PDTS_ bits**. *lpvBuffer* is unused and ignored.
GPA_RATE	Retrieves the sampling rate in samples per second. *lpvBuffer* is unused and ignored.
GPA_RECTBOUND	Retrieves the bounding rectangle of all pen-down points. *lpvBuffer* is the address of a **RECT** structure.
GPA_RECTBOUNDINK	Like GPA_RECTBOUND, retrieves the bounding rectangle of all pen-down points, but inflates the rectangle to accommodate ink width. *lpvBuffer* is the address of a **RECT** structure.
GPA_SIZE	Retrieves the size of the pen data memory block in bytes. Because of the potential large size of this value, the return value of the function is not used. Instead, *lpvBuffer* is the address of a DWORD variable to fill with the size.
GPA_STROKES	Retrieves the total number of strokes, including pen-up strokes. *lpvBuffer* is unused and ignored.
GPA_TIME	Retrieves the absolute time of creation of the pen data. *lpvBuffer* is the address of an **ABSTIME** structure.
GPA_USER	Retrieves the number of user bytes

	available per stroke: 0, 1, 2, or 4. *lpvBuffer* is unused and ignored.
GPA_VERSION	Retrieves the version number of the pen data. *lpvBuffer* is unused and ignored.

Return Value

Returns PDR_OK or an integer value if successful, depending on the *uOption* parameter.

Comments

GetPenDataAttributes provides enhancements of some of the capabilities of **GetPenDataInfo**. It also provides additional detailed information taken from the **HPENDATA** block.

See Also

GetStrokeAttributes, **GetPenDataInfo**

GetPenDataInfo Overview

Overview

1.0 2.0

This function retrieves information from an **HPENDATA** memory block. It is superseded by the **GetPenDataAttributes** function.

BOOL GetPenDataInfo(HPENDATA *hpndt*, **LPPENDATAHEADER** *lppdh*, **LPPENINFO** *lppeninfo*, **DWORD** *dwReserved* **)**

Parameters

hpndt

Handle to a pen data object that receives the pen data information.

lppdh

Address of a **PENDATAHEADER** structure, or NULL if not required.

lppeninfo

Address of a **PENINFO** structure, or NULL if not required.

dwReserved

Reserved for future use. Must be set to 0.

Return Value

Returns TRUE if successful. The return value is FALSE if invalid parameters are used, or if the handle to the pen data is invalid, or if the requested **PENINFO** does not exist in the pen data.

Comments

This function retrieves the header and pen information in the pen data memory block. If *lppeninfo* is not NULL and the pen data does not contain pen information, the contents of *lppeninfo* are not changed. The **wPndts** member in the **PENDATAHEADER** structure can be checked to see if the **HPENDATA** object has a **PENINFO** structure associated with it (a value of PTDS_NOPENINFO indicates not). The amount of data allocated is contained in the **cbSizeUsed** member of the **PENDATAHEADER** structure.

See Also

GetPenDataAttributes

GetPenDataStroke Overview

Overview

1.0 2.0

Returns a pointer to stroke data contained in an **HPENDATA** memory block previously locked with the **BeginEnumStrokes** function.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions. Use other services such as **GetPenDataAttributes**, **GetPointsFromPenData**, or **GetStrokeAttributes** to examine an HPENDATA block.

BOOL GetPenDataStroke(LPPENDATA *lppd*, **UINT** *iStrk*, **LPPOINT FAR *** *lplppt*, **LPVOID FAR *** *lplpvOem*, **LPSTROKEINFO** *lpsi* **)**

Parameters

lppd

Address of the **HPENDATA** memory block. This parameter is the value returned by a previous call to the **BeginEnumStrokes** function.

iStrk

Zero-based index of the stroke to retrieve.

lplppt

Address of a pointer to a point. The pointer returned by the function will point to the first point of the stroke inside the pen data object. This parameter can be NULL if point data is not required.

lplpvOem

Address of a void pointer. The pointer returned by the function will point to the OEM data block of the stroke inside the pen data object. The format of the OEM data is specified by the **rgoempeninfo** member in the **PENINFO** structure. This parameter can be NULL if OEM data is not required.

lpsi

Address of a **STROKEINFO** structure. This parameter can be NULL if stroke information is not required.

Return Value

Returns TRUE if successful. If the stroke requested is out of range, the function returns FALSE.

Comments

GetPenDataStroke returns in *lpsi* a pointer to a **STROKEINFO** structure created from the stroke referenced by *iStrk*. The *lpsi* parameter does not point directly into the **HPENDATA** memory block.

However, the *lplppt* argument points to the first point of the stroke inside the **HPENDATA** block. For a description of how the **GetPenDataStroke** function has changed in version 2.0 of the Pen API, refer to AppendixA.

Applications must call **BeginEnumStrokes** before calling **GetPenDataStroke**. After the last call to **GetPenDataStroke**, the application must call **EndEnumStrokes**. Once **EndEnumStrokes** is called, the data that *lplppt* and *lplpvOem* point to is no longer valid.

Under no circumstances should an application modify data directly within an **HPENDATA** block. Doing so

can invalidate other information in the block. To modify an **HPENDATA** block, use one of the Pen API functions listed in Chapter 4, "The Inking Process."

See Also

BeginEnumStrokes, **EndEnumStrokes**, **GetStrokeAttributes**

GetPenHwEventData Overview

Overview

1.0 2.0

Gets the pen data associated with events in a given range.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions. Use **DoDefaultPenInput** or **GetPenInput** instead.

REC GetPenHwEventData(UINT *wEventRefBeg*, **UINT** *wEventRefEnd*, **LPPOINT** *lppt*, **LPVOID** *lpvOemData*, **int** *cPntMax*, **LPSTROKEINFO** *lpsi* **)**

Parameters

wEventRefBeg

Beginning pen event.

wEventRefEnd

Ending pen event.

lppt

Address of a an array of **POINT** structures. The size of the array must be at least sizeof(POINT) multiplied by *cPntMax*.

lpvOemData

Buffer to fill with OEM-specific data. This can be NULL if no data is required.

cPntMax

Maximum number of samples to return.

lpsi

Address of a **STROKEINFO** structure that receives the stroke information, including the count of points and point state. Also included is the time stamp of the first point returned in the buffer, which is the number of milliseconds that have elapsed since Windows started.

Return Value

Returns REC_OK if successful; otherwise, the return value can be one of the following:

Constant	Description
REC_BUFFERTOOSMALL	The array identified by *lppt* is not large enough to hold all the points requested.
REC_PARAMERROR	Invalid parameter.

Comments

This function fetches all data collected from the pen event *wEventRefBeg* up to but not including the pen event *wEventRefEnd*. If *wEventRefBeg* equals *wEventRefEnd*, **GetPenHwEventData** retrieves the single pen event associated with *wEventRefBeg*.

The values for *wEventRefBeg* and *wEventRefEnd* are obtained by calling the Windows **GetMessageExtraInfo** function.

This function can be called directly from an application. If it returns REC_BUFFERTOOSMALL, no data is returned and the **cPnt** member of *lpsi* contains the number of points between *wEventRefBeg* and *wEventRefEnd*. If REC_OK is returned, the **cPnt** member contains the number of valid points placed in the array at *lppt*.

See Also

STROKEINFO

GetPenInput Overview

Overview

2.0

Collects data after **StartPenInput** has started pen input.

int GetPenInput(HPCM *hpcm*, **LPPOINT** *lppt*, **LPVOID** *lpvOem*, **UINT** *fuOemFlags*, **UINT** *cPntMax*, **LPSTROKEINFO** *lpsi* **)**

Parameters

hpcm

Handle to the current collection. This is the return value from **StartPenInput**.

lppt

Address of an array of **POINT** structures. The array must consist of at least *cPntMax* structures.

lpvOem

The address of a buffer of OEM data associated with each point. This parameter can be NULL if the application does not require OEM data.

fuOemFlags

Flags specifying which OEM data to retrieve. If this parameter is NULL, all OEM data provided by the tablet is returned in the order specified by the **rgoempeninfo** array in **PENINFO**.

These flags have an implicit order. For example, if pressure and barrel rotation are specified, *cPntMax* pairs of these data are returned, in the order [pressure, rotation], [pressure, rotation], and so on.

Constant	Description
PHW_PRESSURE	Retrieve pressure data.
PHW_HEIGHT	Retrieve height data.
PHW_ANGLEXY	Retrieve data pertaining to the x- and y-coordinates.
PHW_ANGLEZ	Retrieve data pertaining to the z-coordinates.
PHW_BARRELROTATION	Retrieve barrel-rotation data.
PHW_OEMSPECIFIC	Retrieve OEM-specific data.
PHW_PDK	Retrieve per-point PDK_ information in OEM data.

cPntMax

The number of **POINT** structures in the array at *lppt*. This is the maximum number of points to return and also the maximum number of OEM items the buffer at *lpvOem* can hold.

lpsi

A pointer to a **STROKEINFO** structure. This structure receives information about the first point of the collection of points placed into the array at *lppt*. The **cbPnts** member contains the packet ID of the first point. All returned points in the collection have the same tip polarity (that is, up or down) as the first point.

Return Value

Returns 0 if there are no points available. If the return value is positive, the value is the number of points

copied to the *lppt* (and, optionally, *lpvOem*) buffers. Otherwise, the return value is one of the following:

Constant	Description
PCMR_APPTERMINATED	Input has already terminated because the application called **StopPenInput**. There are no more points to retrieve.
PCMR_EVENTLOCK	An event must be taken out of the queue using the Windows functions **PeekMessage** or **GetMessage** before any more points can be retrieved using **GetPenInput**.
PCMR_INVALIDCOLLECTION	The *hpcm* handle is invalid because the calling application did not start input with **StartPenInput**.
PCMR_TERMTIMEOUT	Input has already terminated because the specified time-out period has elapsed.
PCMR_TERMRANGE	Input has already terminated because the pen has left the range of the tablet's zone of sensitivity.
PCMR_TERMPENUP	Input has already terminated because the pen was lifted from the tablet.
PCMR_TERMEX	Input has already terminated because the pen went down in a specified exclusion rectangle or region.
PCMR_TERMBOUND	Input has already terminated because the pen went down outside a specified bounding rectangle or region.

Comments

Once an application initiates pen-input collection by calling **StartPenInput**, the application then calls the **GetPenInput** function frequently to retrieve the actual data arriving from the pen device. This can be done by responding to hardware events or by continuously polling.

In the polling model, the application repeatedly calls **GetPenInput** to get data. It is important for the application to yield periodically; for example, by calling **PeekMessage**. A fast loop can potentially process the points before the system can collect more. In this case, successive calls to **GetPenInput** return 0 until the user writes some more. Polling is typically terminated when **GetPenInput** detects and returns a termination condition specified in **StartPenInput**.

In the event model, the application calls **GetPenInput** on receipt of a WM_PENEVENT message. All points up to this event are returned to the caller. An application can retrieve all available data in a short loop, until **GetPenInput** returns PCMR_EVENTLOCK. The application then falls out of the loop and exits the window procedure. The process begins again when the window procedure is called in response to another WM_PENEVENT message in the application's message queue.

If *lpvOem* is not NULL, the buffer must be large enough to hold *cPntMax* OEM data packets. The size of each packet is the width specified in the **cbOemData** member of the **PENINFO** structure, plus sizeof(UINT) if PDK_ values are required.

Example

The following code example gathers more pen input for use by the recognizer. Assume the application has already called **StartPenInput** and is using the messaging collection model.

```
POINT      rgPnt[cbBuffer];
STROKEINFO si;

// ... in WM_PENEVENT message handler:

switch (wParam)
{
      .
      .
      .
case PE_PENUP:
case PE_PENMOVE:
case PE_TERMINATING:

      // Get all the points collected since the last message

      while ( (iRet = GetPenInput( hpcm, rgPnt, NULL, 0,
                                                   cbBuffer, &si) ) > 0)
      {

            // Add pen data to recognition context and def process

            AddPenInputHRC( vhrc, rgPnt, NULL, 0, &si );
            ProcessHRC( vhrc, PH_DEFAULT );
      }
      break;
```

See Also

PeekPenInput, **StartPenInput**, PDK_

GetPenResource Overview

Overview

2.0

The **GetPenResource** function retrieves a copy of a pen services resource. (Japanese version only.)

HANDLE GetPenResource(WPARAM *wParam* **)**

Parameters

wParam

Specifies the pen services resource for which to retrieve a handle. This may be one of the following:

Constant	Description
GPR_CURSPEN	Standard pen cursor.
GPR_CURSCOPY	Copy cursor.
GPR_CURSUNKNOWN	Unknown cursor.
GPR_CURSERASE	Erase cursor.
GPR_BMCRMONO	Monochrome Return bitmap.
GPR_BMLFMONO	Monochrome LineFeed bitmap.
GPR_BMTABMONO	Monochrome Tab bitmap.
GPR_BMDELETE	Delete bitmap.
GPR_BMLENSBTN	Lens buttonface bitmap.
GPR_BMHSPMONO	Hankaku space bitmap (Japanese version only).
GPR_BMZSPMONO	Zenkaku space bitmap (Japanese version only).

Comments

An application can use this function to get a copy of a cursor or bitmap used by pen services. It is the application's responsibility to destroy the object by calling either the **DestroyCursor** or **DeleteObject** Windows API.

Return Value

This function returns a handle to an object, depending on the index specified by *wParam* if successful. Otherwise the return value is NULL.

GetPenMiscInfo Overview

Overview

1.0 2.0

Retrieves values pertaining to the pen system.

LONG GetPenMiscInfo(WPARAM *wParam*, **LPARAM** *lParam* **)**

Parameters

wParam

Specifies the identifier of the pen system value to retrieve. The pen system identifier must be a PMI_ value. See the table below for the possible PMI_ values in *wParam*.

lParam

Address of storage for data. This must not be NULL. The calling application must ensure that there is sufficient room to store the requested information. The type of storage object that *lParam* points to depends on *wParam*, as described in the following table. For each value of *wParam* in the first column, the second column describes the corresponding requirement for *lParam*:

wParam constant	LParam description
PMI_BEDIT	*lParam* is the address of a **BOXEDITINFO** structure. Boxed edit information.
PMI_CXTABLET	*lParam* is a far pointer to a UINT value specifying the width of tablet (in units of 0.001 inch) if present; otherwise, the width of the screen.
PMI_CYTABLET	*lParam* is a far pointer to a UINT value specifying the height of tablet (in units of 0.001 inch) if present; otherwise, the height of the screen.
PMI_INDEXFROMRGB	*lParam* is a far pointer to a DWORD value. On entry, *lParam* is the address of an **RGB** ink color value. On return, the low-order word of *lParam* is replaced with an index in the range 0 to 15 for the closest standard ink color and the high-order word is 0.
PMI_ENABLEFLAGS	*lParam* is a far pointer to a WORD value containing a flag describing whether certain Pen API features are enabled. The flags can be a combination of the following values: PWE_AUTOWRITE Enable pen functionality where the I-Beam cursor is present. PWE_ACTIONHANDLES Enable action handles in controls. PWE_INPUTCURSOR Show cursor while writing. PWE_LENS Enable pop-up letter guides (that is, the lens).
PMI_PENTIP	*lParam* is the address of a **PENTIP** structure.

PMI_RGBFROMINDEX	*lParam* is the address of a DWORD value. On entry, *lParam* is the address of an index in the range 0 to 15; on return, this value at this address is replaced with the standard **RGB** ink color value.
wParam constant	**LParam description**
PMI_SYSFLAGS	*lParam* is a far pointer to a WORD value containing a flag describing which pen system components are loaded. The flags can be a combination of the following values: PWF_RC1 Support available for Pen API version 1.0 Recognition Context (RC) and associated functions. PWF_PEN Pen/tablet hardware is present. PWF_INKDISPLAY Ink-compatible display driver is present. PWF_RECOGNIZER System recognizer is present. PWF_BEDIT Boxed edit (bedit) control is available. PWF_HEDIT Handwriting edit (hedit) control is available. PWF_IEDIT Ink edit (iedit) control is available. PWF_ENHANCED Enhanced features, including gesture support and 1 millisecond timing, are available. PWF_FULL All components listed above are present..
PMI_SYSREC	*lParam* is a far pointer to an **HREC** value which is the handle of the system recognizer, if present.
PMI_TICKREF	*lParam* is the address of an **ABSTIME** structure indicating the absolute reference time that the system uses to calculate time-stamps for strokes in pen data objects and inkset
PMI_TIMEOUT	*lParam* is a far pointer to a UINT value indicating time-out value to end hand-writing input, in milliseconds.
PMI_TIMEOUTGEST	*lParam* is a far pointer to a UINT value indicating time-out value to end a gesture, in milliseconds.
PMI_TIMEOUTSEL	*lParam* is a far pointer to a UINT value indicating the time-out value in milli-seconds for press-and-hold gesture. The range of permissible values is 0 to 5000. If press-and-hold has been disabled, this value is 65,535.

Return Value

The return value is PMIR_OK if successful; otherwise it is one of the following negative error values:

Constant	Description
PMIR_INDEX	*wParam* is out of range.
PMIR_NA	Support for this value of *wParam* is not available.
PMIR_VALUE	*lParam* is NULL or a invalid pointer.

Comments

The information type returned varies depending on the index. Note that if a UINT is expected, for example, it is an error to provide the address of a DWORD variable without explicitly setting the HIWORD to 0. This function only sets the LOWORD in this case, and since the variable is usually declared on the stack, there would be an unknown value in the HIWORD. See the examples below.

If *wParam* is PMI_INDEXFROMRGB or PMI_RGBFROMINDEX, the standard pen-tip color table is as follows:

```
00  black       RGB(  0,   0,   0)
01  dark blue   RGB(  0,   0, 127)
02  dark green  RGB(  0, 127,   0)
03  dark cyan   RGB(  0, 127, 127)
04  dark red    RGB(127,   0,   0)
05  purple      RGB(127,   0, 127)
06  brown       RGB(127, 127,   0)
07  gray        RGB(127, 127, 127)
08  light gray  RGB(192, 192, 192)
09  blue        RGB(  0,   0, 255)
10  green       RGB(  0, 255,   0)
11  cyan        RGB(  0, 255, 255)
12  red         RGB(255,   0,   0)
13  magenta     RGB(255,   0, 255)
14  yellow      RGB(255, 255,   0)
15  white       RGB(255, 255, 255)
```

Example

The following code sample retrieves the timeout and pen tip:

```
UINT uTimeout;
PENTIP tip;

GetPenMiscInfo( PMI_TIMEOUT, (LPARAM)(UINT FAR *)&utimeout );
GetPenMiscInfo( PMI_PENTIP, (LPARAM)(LPPENTIP)&tip );
```

Note that the following is an error, since the HIWORD is undefined:

```
DWORD dwTimeout;

GetPenMiscInfo( PMI_TIMEOUT, (LPARAM)&dwtimeout );     // Wrong!
```

See Also

SetPenMiscInfo, PMI_

GetPointsFromPenData Overview

Overview

1.0 2.0

Retrieves a specified range of points.

BOOL GetPointsFromPenData(HPENDATA *hpndt*, **UINT** *iStrk*, **UINT** *iPnt*, **UINT** *cPnts*, **LPPOINT** *lppt* **)**

Parameters

hpndt

Handle to a pen data object.

iStrk

The zero-based stroke index from which points are retrieved.

iPnt

First point to retrieve from the specified stroke.

cPnts

Number of points to retrieve. If this value is 0, the function returns TRUE.

lppt

Address of buffer to fill with points.

Return Value

Returns TRUE if successful, or FALSE if the requested points are out of range.

Comments

GetPointsFromPenData performs a function similar to **GetPenDataStroke** in that it retrieves information from an **HPENDATA** memory block. But **GetPointsFromPenData** copies the required data to buffers supplied by the application, rather than simply returning pointers to the original data in the global heap.

An application can also request a copy of a particular subset of points within a stroke. In this case, *iPnt* identifies the first point and *cPnts* is the number of points to retrieve. This allows an application to digest the points in an **HPENDATA** block a few at a time to avoid having to allocate a large block of memory for the entire set of points.

GetPointsFromPenData returns the last point in a stroke if the *iPnt* argument is set to a value larger than the total number of points in the stroke. In the same manner, the function returns the points of the last stroke if *iStrk* exceeds the total number of strokes in the **HPENDATA** block. If the count of points to return is 1 and *iPnt* is beyond the last point in the stroke, the function returns the last point in the stroke.

See Also

GetPenDataStroke

GetResultsHRC Overview

Overview

2.0

Retrieves results from a recognition context **HRC**. A recognizer must export this function.

int GetResultsHRC(HRC *hrc*, **UINT** *uType*, **LPHRCRESULT** *rghrcresults*, **UINT** *cResults* **)**

Parameters

hrc

Handle to the **HRC** object.

uType

Specifies the type of expected results. This can be one of the following values:

Constant	Description
GRH_ALL	Return all results.
GRH_GESTURE	Return results of type gesture only.
GRH_NONGESTURE	Return all results not of type gesture.

rghrcresults

Address of an array of **HRCRESULT** objects.

cResults

The size of the *rghrcresults* array, in objects. The actual size in bytes can be calculated by multiplying *cResults* by the size of **HRCRESULT**. This parameter must be greater than 0.

Return Value

Returns the actual number of results returned if successful. This can be 0; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_HOOKED	A hook preempted the result.
HRCR_MEMERR	Insufficient memory.

Comments

The actual number of results returned by this function may be less than the number specified by *cResults*. It is also less than or equal to the count specified at the creation of *hrc* by the *cMaxResults* parameter in **SetMaxResultsHRC**, regardless of the size of the *rghrcresults* array.

A return value of 0 indicates that the recognizer was not able to recognize any of the input, even if coerced by a word list set into the **HRC**. A recognizer should never return a result consisting entirely of SYV_UNKNOWN symbols.

The calling application must explicitly destroy each valid result using **DestroyHRCRESULT**. However, if the return value is 0 or negative, the contents of the *rghrcresults* array are undefined (though not NULL) and **DestroyHRCRESULT** must not be called.

See Also

DestroyHRCRESULT

GetStrokeAttributes Overview

Overview

2.0

Retrieves information about a stroke in an **HPENDATA** object.

int GetStrokeAttributes(HPENDATA *hpndt*, **UINT** *iStrk*, **LPVOID** *lpvBuffer*, **UINT** *uOption* **)**

Parameters

hpndt

Handle to the **HPENDATA** object, which must not be compressed.

iStrk

Zero-based stroke index. If there are no strokes in the pen data, an index of 0 can be used to retrieve the default attributes for the pen data. A value of IX_END specifies the last available stroke in the pen data.

lpvBuffer

Pointer to a structure whose type depends on *uOption*, or NULL if the *uOption* parameter does not require this buffer.

uOption

Specifies the attributes to retrieve. This parameter has one of the following values:

Constant	Description
GSA_DOWN	Retrieve the up/down state of the pen tip for this stroke. Returns 1 if the stroke is a down-stroke or 0 if it is an up-stroke. *lpvBuffer* is unused and ignored.
GSA_PENTIP	Retrieve the pen-tip characteristics (color, width, nib) used by the stroke specified by *iStrk*. *lpvBuffer* is a pointer to a **PENTIP** structure. Return value is PDR_OK.
GSA_PENTIPCLASS	Retrieve the pen-tip characteristics (color, width, nib), if any, for the class of strokes of which the stroke specified by *iStrk* is a member. *lpvBuffer* is a pointer to a **PENTIP** structure. Return value is PDR_OK.
GSA_RECTBOUND	Retrieve the bounding rectangle of the specified stroke. *lpvBuffer* is a pointer to a **RECT** structure. Return value is PDR_OK.
GSA_SELECT	Retrieve the selection status of the specified stroke. *lpvBuffer* is unused and ignored. Returns a nonzero value if the stroke is selected; otherwise, the return value is 0.
GSA_SIZE	Retrieve size of stroke in points and bytes. *lpvBuffer* is a pointer to a double-word value, or NULL. LOWORD(*(LPDWORD)*lpvBuffer*) is the size in points, and

	HIWORD(*(LPDWORD)*lpvBuffer*) is the size in bytes. Return value is PDR_OK.
GSA_TIME	Retrieve the absolute time of the stroke. *lpvBuffer* is a pointer to an **ABSTIME** structure; it cannot be NULL. The **sec** field specifies the number of seconds since Jan 1, 1970, and the **ms** field specifies the number of milliseconds offset from that time to the beginning of the stroke. Return value is PDR_OK.
GSA_USER	Retrieve the user value, if any, for the stroke. *lpvBuffer* is a pointer to a double-word value, or NULL. Returns the number of bytes of user data available in the stroke: 0, 1, 2, or 4.
GSA_USERCLASS	Retrieve the user value, if any, for the class of strokes of which the stroke specified by *iStrk* is a member. *lpvBuffer* is a pointer to a double-word value, or NULL. The return value is 4 because the user value in the strokes class table is a doubleword value.

Return Value

Returns PDR_OK or an integer value if successful, as described for the *uOption* parameter. If an error occurs, returns one of the following:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Memory error.
PDR_PNDTERR	Invalid pen data.
PDR_STRKINDEXERR	Invalid stroke index.
PDR_TIMESTAMPERR	Timing information was removed.
PDR_VERSIONERR	Could not convert old pen data.

See Also

CreatePenDataEx, **GetStrokeTableAttributes**, **SetStrokeAttributes**, **SetStrokeTableAttributes**, **PENTIP**

GetStrokeTableAttributes Overview

Overview

2.0

Retrieves information about a stroke's class from the table in the **PENDATAHEADER** of an **HPENDATA** object.

int GetStrokeTableAttributes(HPENDATA *hpndt*, **UINT** *iTblEntry*, **LPVOID** *lpvBuffer*, **UINT** *uOption* **)**

Parameters

hpndt

Handle to the **HPENDATA** object, which must not be compressed.

iTblEntry

Zero-based table index to the class entry in the pen data header.

lpvBuffer

Pointer to a structure whose type depends on *uOption*, or NULL if the *uOption* parameter does not require this buffer.

uOption

Specifies the attributes to retrieve. This parameter can be one of the following:

Constant	Description
GSA_PENTIPTABLE	Retrieve the pen-tip characteristics (color, width, nib) of the class of strokes specified by *iTblEntry*. *lpvBuffer* is a pointer to a **PENTIP** structure. Return value is the number of strokes using this class.
GSA_SIZETABLE	Retrieve the number of entries in the stroke class table. *iTblEntry* and *lpvBuffer* are unused and ignored. Return value is the number of classes used in the stroke class table.
GSA_USERTABLE	Retrieve the user value, if any, of the class of strokes specified by *iTblEntry*. *lpvBuffer* is a pointer to a doubleword value, or NULL. The number of bytes that are valid in *lpvBuffer* depends on flags set in **CreatePenDataEx**. This number is returned by the function, and can be 0 (no user value), 1 (byte value), 2 (word value), or 4 (doubleword value). Return value is 4, because the user value in the stroke class table is a doubleword value.

Return Value

Returns an integer if successful, depending on the value of *uOption*, as described above. If an error occurs, returns one of the following:

Constant	Description

PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Memory error.
PDR_PNDTERR	Invalid pen data.
PDR_VERSIONERR	Could not convert old pen data.

See Also

CreatePenDataEx, **GetStrokeAttributes**, **SetStrokeAttributes**, **SetStrokeTableAttributes**, **PENTIP**

GetSymbolCount Overview

Overview

1.0 2.0

Returns the number of symbol strings contained in a symbol graph **SYG**.

int GetSymbolCount(LPSYG *lpsyg* **)**

Parameters

lpsyg

Address of the symbol graph.

Return Value

Returns the number of possible symbol strings that can be generated from the symbol graph. Returns -1 for any graph that can generate more than 32,767 symbol strings, or if there is a parameter error.

Example

For example, if the symbol graph pointed to by *lpsyg* is

```
ex {a | u} mple
```

GetSymbolCount returns the value 2 because the graph contains two symbol strings ("example" and "exumple").

See Also

EnumSymbols, **FirstSymbolFromGraph**, **GetSymbolMaxLength** **SYG**, SYV_

GetSymbolCountHRCRESULT Overview

Overview

2.0

Retrieves the count of symbols available in a recognition result. A recognizer must export this function.

int GetSymbolCountHRCRESULT(HRCRESULT *hrcresult* **)**

Parameters

hrcresult

Handle of a results object.

Return Value

Returns the count of symbols if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.

Comments

This function is typically called before **GetSymbolsHRCRESULT** to determine the size of a buffer required to store the symbol values returned in a recognition result. To calculate the size of the buffer, multiply the value returned by this function by sizeof(SYV).

See Also

GetSymbolsHRCRESULT, SYV_

GetSymbolMaxLength Overview

Overview

1.0 2.0

Returns the length of the longest symbol string contained in a symbol graph **SYG**.

int GetSymbolMaxLength(LPSYG *lpsyg* **)**

Parameters

lpsyg

Address of the symbol graph.

Return Value

Returns the number of symbols in the longest symbol string that can be generated from the symbol graph, or -1 if there is a parameter error.

Example

For example, if the symbol graph pointed to by *lpsyg* is

```
ab {c | de} f
```

GetSymbolMaxLength returns 5 because the longest string is "abdef".

See Also

EnumSymbols, **FirstSymbolFromGraph**, **SYG**, SYV_

GetSymbolsHRCRESULT Overview

Overview

2.0

Retrieves an array of symbol values corresponding to a recognition result. A recognizer must export this function.

int GetSymbolsHRCRESULT(HRCRESULT *hrcresult*, **UINT** *iSyv*, **LPSYV** *rgsyv*, **UINT** *cSyv* **)**

Parameters

hrcresult

Handle of a results object.

iSyv

Index of the first symbol of interest in the results object.

rgsyv

Address of a buffer in which to put the symbols. The array must be large enough to store *cSyv* symbols.

cSyv

The size of *rgsyv* in symbols (not bytes). This is the number of symbols to be returned. A value of 0 is legal, in which case the function simply returns 0.

Return Value

Returns the count of symbols copied, if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.

Comments

It is possible to allocate a small buffer in *rgsyv* and call this function repeatedly, incrementing the index *iSyv* each time by the number of symbols returned in the previous call, until the function returns 0.

Example

The following example gets a character result, using a small buffer:

```
#define CBCHBUF 1024   // Char buffer
#define CSYVMAX 32     // Relatively small symbol chunk

HRC    vhrc;           // Handle to a handwriting context
HRCRESULT vhrcresult;        // Handle to a recognition result
SYV    vrgsyv[CSYVMAX];    // Symbol result buffer
char   vrgcBuff[CBCHBUF];  // Buffer for recognition results
     .
     .          // Code that creates HRC, gets input, etc....
     .
EndPenInputHRC( vhrc );     // Tell recognizer no more ink
ProcessHRC( vhrc, PH_MAX ); // Finish recognition
```

```
      .
      .          // Retrieve a handle to the results
      .
if (GetResultsHRC( vhrc, &vhrcresult, 1 ) > 0)
{
      int i = 0, cSyv;

      // Retrieve some symbols
      while ((cSyv = GetSymbolsHRCRESULT( vhrcresult,
            i, vrgsyv, CSYVMAX )) > 0)
      {
      if (i + cSyv + 1 > CBCHBUF) // Don't overflow buffer
            cSyv = CBCHBUF - i - 1;
      if (cSyv > 0)               // Still have something?
      {
            SymbolToCharacter( vrgsyv, cSyv, vrgcBuff + i, NULL );
            i += cSyv;
      }
      if (i + 1 >= CBCHBUF)
            break;
      }
      vrgcBuff[i] = chNull;    // Terminate string
}

DestroyHRCRESULT( vhrcresult );  // We're finished with result
vhrcresult = NULL;
DestroyHRC( vhrc );           // Finished with this HRC session
vhrc = NULL;
```

See Also

GetSymbolCountHRCRESULT, SYV_

GetVersionPenWin Overview

Overview

1.0 2.0

Retrieves the Pen API version number.

UINT GetVersionPenWin( )

Return Value

The low-order byte of the return value specifies the major (version) number. The high-order byte specifies the minor (revision) number.

GetWordlistCoercionHRC Overview

Overview

2.0

Retrieves the current word list coercion setting in a handwriting-recognition context **HRC**.

int GetWordlistCoercionHRC(HRC *hrc* **)**

Parameters

hrc

Handle to the **HRC** object.

Return Value

If successful, returns one of the following values:

Constant	Description
SCH_ADVISE	The word list is a hint to the recognizer, and results are not strongly coerced to match the word list.
SCH_FORCE	If results do not match the word list, the closest fit is returned.
SCH_NONE	Do not coerce. This flag can be used to turn off a previous request.

Otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

See Also

SetWordlistCoercionHRC

GetWordlistHRC Overview

Overview

2.0

Retrieves a word list from a recognition context **HRC**.

int GetWordlistHRC(HRC *hrc*, **LPHWL** *lphwl* **)**

Parameters

hrc

Handle to the **HRC** object.

lphwl

Address of a handle to a word list. The recognizer sets the handle to NULL if the recognition context does not contain a word list.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

An **HRC** can be configured for only one word list at a time. This is independent of the recognizer's dictionary, which can be manipulated through the **EnableSystemDictionaryHRC** function.

For a description of word lists and how a recognizer uses them, see "Configuring the HRC" in Chapter 5, "The Recognition Process."

See Also

SetWordlistHRC

HitTestPenData Overview

Overview

2.0

Determines if a given point lies on or near the pen-down strokes contained in an **HPENDATA** object.

int HitTestPenData(HPENDATA *hpndt*, **LPPOINT** *lppt*, **UINT** *dThresh*, **UINT FAR*** *lpiStrk*, **UINT FAR*** *lpiPnt* **)**

Parameters

hpndt

Handle to the **HPENDATA** object. **HitTestPenData** does not alter the data in the **HPENDATA** object.

lppt

Address of a **POINT** structure containing the point to test.

dThresh

Threshold around the point given in *lppt*. The point lies at the center of a square with sides of length *dThresh*. If **HitTestPenData** finds a point in the **HPENDATA** object that lies in the square, it indicates a "hit." *dThresh* must have the same scaling units as the points in the **HPENDATA** or the result will not be correct. If *dThresh* is 0, **HitTestPenData** assumes a default threshold value of 3.

lpiStrk

Stroke index from which to begin testing. After **HitTestPenData** returns from a successful test, the variable that *lpiStrk* points to contains the index of the hit stroke.

lpiPnt

Point index from which to begin testing. After **HitTestPenData** returns from a successful test, the variable that *lpiPnt* points to contains the index of the hit point in the stroke indicated by *lpiStrk*.

Return Value

Returns one of the following if successful:

Constant	Description
PDR_HIT	The point hits (intersects) the pen data or falls within the provided threshold around a particular point in the pen data as specified by the stroke and the point indices. The stroke and point values are returned in *lpiStrk* and *lpiPnt*, respectively.
PDR_NOHIT	The point does not hit (intersect) the pen data nor does it fall within the provided threshold around a particular point in the pen data as specified by the stroke and the point indices.

Otherwise, the function returns one of the following negative values:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Memory error.
PDR_PNDTERR	Invalid pen data.

PDR_PNTINDEXERR	Invalid point index.
PDR_STRKINDEXERR	Invalid stroke index.
PDR_VERSIONERR	Could not convert old pen data.

Comments

HitTestPenData checks whether the point specified by *lppt* falls within the threshold specified by *dThresh* around a point in the pen data, depending on the zero-based starting stroke and point indices specified by *lpiStrk* and *lpiPnt*. This function tests only down strokes in the pen data. If *lpiStrk* is greater than the number of strokes in the pen data, testing starts from the first stroke.

Similarly, if *lpiPnt* is greater than the number of points in the stroke, testing starts from the first point in that stroke. The first point in the first stroke (from the specified indices) that meets the test condition is returned via *lpiPnt* and *lpiStrk*, respectively.

In any case, **HitTestPenData** accounts for the width of the ink trail. If the value given in *dThresh* is less than the ink width, **HitTestPenData** ignores the specified value of *dThresh* and instead uses the ink width as the threshold.

HitTestPenData does not consider pen-up strokes.

InitRC Overview

Overview

1.0 2.0

Initializes an RC structure with default values.

Note This function is provided only for compatibility with version 1.0 of the Pen API, and will not be supported in future versions.

void InitRC(HWND *hwnd*, **LPRC** *lprc* **)**

Parameters

hwnd

Handle to a window.

lprc

Address of the RC structure to initialize.

Return Value

This function does not return a value.

Comments

InitRC serves little purpose in applications that conform to version 2.0 of the Pen API. Under version 2.0, a recognizer maintains an **HRC** object, which makes the **RC** structure obsolete.

For suggestions on how to update a version 1.0 application to remove services that rely on **RC**, see the section "The RC Structure" in Appendix A.

InitRC initializes an **RC** structure with default values, many of which come from the global **RC** structure. The application can use the initialized **RC** structure when calling the **Recognize** function. Although an application can change any of these values, it should be careful about changing those items that can be set by the user through the Windows Control Panel.

InitRC sets the bounding rectangle to the client area of the window identified by *hwnd*. The bounding rectangle is valid only until the window is resized or moved. When this occurs, the application must again call **InitRC** to update the **rectBound** member of the RC structure or correct **rectBound** manually. If the window handle *hwnd* is NULL, the bounding rectangle and **hwnd** remain uninitialized. The application must set the **hwnd** member to a valid window before calling **Recognize** or **RecognizeData**.

The following table describes the default values used to initialize the **RC** structure. Values not listed in the table come from the global **RC**. Some of the global default values can be modified by the user in Control Panel.

Value	Description
rc.alc	ALC_DEFAULT. The function uses the complete alphabet and all gestures. The exact character set depends on the recognizer.
rc.lRcOptions	Zero.
rc.hwnd	*hwnd* (the first argument of **InitRC**).

rc.wResultMode	RRM_COMPLETE.
rc.rectBound	(0,0,0,0) or client rectangle of *hwnd* if *hwnd* is not NULL.
rc.lPcm	PCM_ADDDEFAULTS, or PCM_ADDDEFAULTS | PCM_RECTBOUND if *hwnd* is not NULL.
rc.rectExclude	(0,0,0,0).
rc.guide	(0,0,0,0,0,0,0).
rc.wRcOrient	RCOR_NORMAL.
rc.wRcDirect	0x0103

Members the user can change through the system Control Panel are filled with values indicating that the system default should be used. These placeholder values are RC_WDEFAULT or RC_LDEFAULT, depending on whether the member is a UINT or LONG value. During the processing of **ProcessWriting**, **Recognize**, or **RecognizeData**, these values are replaced with the current system defaults before the **RC** structure is passed to the recognizer. If the PCM_ADDDEFAULTS flag is set in **lPcm**, the values of the **lPcm** member in the global **RC** are combined with the current **lPcm** values with OR operators at the time the recognizer is called. If the high bit is set in **wRcPreferences**, the values of the **wRcPreferences** member in the global **RC** are combined with the current **wRcPreferences** values with bitwise-OR operators at the time the recognizer is called.

The following table gives the default values for the members of the **RC** structure:

Value	**Description**
rc.hrec	RC_WDEFAULT
rc.lpfnYield	RC_LDEFAULT
rc.lpUser	RC_LDEFAULT
rc.wCountry	RC_WDEFAULT
rc.wIntlPreferences	RC_WDEFAULTFLAGS
rc.lpLanguage	RC_LDEFAULT
rc.rglpdf	RC_LDEFAULT
rc.wTryDictionary	RC_WDEFAULT
rc.clErrorLevel	RC_WDEFAULT
rc.wTimeOut	RC_WDEFAULT
rc.wRcPreferences	RC_WDEFAULTFLAGS
rc.nInkWidth	RC_WDEFAULT
rc.rgbInk	RC_LDEFAULT
rc.alcPriority	ALC_NOPRIORITY
rc.rgbfAlc	Array initialized to 0

The **RC** structure pointed to in the **RCRESULT** structure is a copy of the original **RC** structure passed as a parameter to **Recognize**. In the copy, default values are replaced. All coordinates are in the tablet coordinate system and the **IRcOptions** member has the RCO_TABLETCOORD flag set.

See Also

Recognize, **RecognizeData**, **RC**, ALC_, PCM_, RCO_

InsertPenData Overview

Overview

2.0

Merges two blocks of pen data at the specified stroke index.

int InsertPenData(HPENDATA *hpndtDst*, **HPENDATA** *hpndtSrc*, **UINT** *iStrk* **)**

Parameters

hpndtDst

Handle of the pen data object to merge into. When this function returns, this is the handle of the merged data.

hpndtSrc

Handle of the pen data object to merge from.

iStrk

Stroke index. The merge operation occurs before this index. This parameter can also be IX_END to append *hpndtSrc* to the end of *hpndtDst*.

Return Value

Returns PDR_OK if successful; otherwise, the return value is one of the following negative values:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Out of memory.
PDR_OEMDATAERR	Incompatible OEM data in the two pen data objects.
PDR_STRKINDEXERR	Invalid stroke index.
PDR_TIMESTAMPERR	Incompatible time fields in the two pen data objects.
PDR_USERDATAERR	Incompatible user space in the two pen data objects.
PDR_VERSIONERR	Could not convert old pen data object.

Comments

InsertPenData merges two blocks of pen data starting at the zero-based stroke index specified by *iStrk* of the destination pen data.

The blocks of pen data to be merged must be compatible. The calling application should ensure that the blocks of data are in the same scaling mode. The user space, if present, should be of the same size. OEM data, if present, must be compatible and of the same type. The application can use **TrimPenData** to delete certain information from either the source or the destination **HPENDATA** object to make it compatible. If *hpndtDst* has timing information and *hpndtSrc* does not, the merge fails. However, if *hpndtDst* does not have timing information and *hpndtSrc* does have it, the timing is stripped from *hpndtSrc*.

For a description of timing information, see "The HINKSET Object" in Chapter 4, "The Inking Process."

See Also

InsertPenDataStroke, InsertPenDataPoints, MetricScalePenData, TrimPenData, CreatePenDataEx

InsertPenDataPoints Overview

Overview

2.0

Inserts points into an existing **HPENDATA** object.

int InsertPenDataPoints(HPENDATA *hpndt*, **UINT** *iStrk*, **UINT** *iPnt*, **UINT** *cPnts*, **LPPOINT** *lppt*, **LPVOID** *lpvOem* **)**

Parameters

hpndt

Handle to an **HPENDATA** object.

iStrk

Zero-based index of the stroke into which the points are inserted. If this value is IX_END, the points are inserted in the last stroke.

iPnt

Zero-based index of the point in the stroke before which the points are inserted. If this value is IX_END, the points are appended to the end of the stroke.

cPnts

Total number of points to be inserted. If this is 0, the function returns PDR_OK without taking any other action.

lppt

Address of an array of **POINT** structures containing the points to be inserted.

lpvOem

Address of a buffer containing the OEM data to be inserted. This value can be NULL only if the **HPENDATA** object does not have OEM data or a **PENINFO** structure.

Return Value

Returns PDR_OK if successful; otherwise, the return value can be one of the following negative values:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Out of memory.
PDR_STRKINDEXERR	Invalid stroke index.
PDR_PNTINDEXERR	Invalid point index.
PDR_VERSIONERR	Could not convert old pen data object.

Comments

InsertPenDataPoints inserts points into an existing stroke of the specified pen data object. It does not create a new stroke. (Use the **InsertPenDataStroke** function to insert new strokes into the pen data object.) The stroke attributes are not affected by the points added to the stroke.

The calling application must ensure that *lppt* and *lpvOem* are valid and that the points are in the same scale as those of the pen data object. **InsertPenDataPoints** performs no automatic scaling of the points.

InsertPenDataPoints does not make any timing adjustments after adding points. This can affect recognition accuracy and should be used judiciously.

For a description of timing information, see "The HINKSET Object" in Chapter 4, "The Inking Process."

See Also

AddPointsPenData, **ExtractPenDataPoints**, **InsertPenData**, **InsertPenDataStroke**, **RemovePenDataStrokes**

InsertPenDataStroke Overview

Overview

2.0

Inserts a stroke into an existing **HPENDATA** object.

int InsertPenDataStroke(HPENDATA *hpndt*, **UINT** *iStrk*, **LPPOINT** *lppt*, **LPVOID** *lpvOem*, **LPSTROKEINFO** *lpsiNew* **)**

Parameters

hpndt

Handle to the **HPENDATA** object that receives the inserted strokes.

iStrk

Zero-based index of the stroke at which the new stroke is to be inserted. If this value is IX_END, the stroke is appended at the end of the **HPENDATA** memory block.

lppt

Pointer to a buffer containing the points to be inserted.

lpvOem

Pointer to a buffer of OEM data. This value can be NULL only if the pen data object does not have OEM data or a **PENINFO** structure.

lpsiNew

Pointer to the **STROKEINFO** structure containing information about the stroke.

Return Value

Returns PDR_OK if successful; otherwise, the return value can be one of the following negative values:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Out of memory.
PDR_STRKINDEXERR	Invalid stroke index.
PDR_TIMESTAMPERR	Timing error.
PDR_VERSIONERR	Could not convert old pen data object.

Comments

InsertPenDataStroke inserts an entire stroke into an **HPENDATA** object. Use **InsertPenDataPoints** to insert points into a particular stroke.

The inserted stroke assumes the default pen-tip attributes. **SetStrokeAttributes** should be called after inserting the stroke to change such stroke attributes as the pen-tip characteristics or user data.

The calling application must ensure that *lppt* and *lpvOem* are valid, and that the points in the stroke that is being added have compatible scaling modes.

Attempting to insert an empty stroke simply returns PDR_OK.

See Also

AddPointsPenData, ExtractPenDataPoints, **InsertPenDataPoints**, RemovePenDataStrokes, STROKEINFO

InstallRecognizer Overview

Overview

1.0 2.0

Loads and initializes a specified recognizer.

HREC InstallRecognizer(LPSTR *lpszRecogName* **)**

Parameters

lpszRecogName

Name of recognizer to load. If *lpszRecogName* is NULL, the default recognizer is loaded. (An application should not set *lpszRecogName* to NULL, because Windows automatically loads the default recognizer on initialization.)

Return Value

Returns a handle to a recognizer if successful; otherwise, returns NULL.

Comments

The recognizer's name is the name of the DLL to be loaded. Windows searches for the recognizer file according to the standard rules for searching for a DLL—that is, it first searches the current directory, then the Windows directory, the system subdirectory, the PATH directories, and so forth. The procedure fails if the library cannot be found, the load fails, or the loaded DLL is not a valid recognizer. The recognizer may decline to load if it requires hardware information such as pen pressure that the pen device cannot provide.

An application should not load the default recognizer. All recognizers installed by an application must be uninstalled by a call to **UninstallRecognizer** before the application terminates.

After loading a recognizer library, the system calls the recognizer's **ConfigRecognizer** function with the subfunction WCR_INITRECOGNIZER.

An application can load a recognizer with a call to **LoadLibrary** instead of **InstallRecognizer**. However, the application must first link with an import library derived from the recognizer DLL. The recognizer's import library must appear in the library section of the LINK command line before PENWIN.LIB. This forces the application's calls to pass directly to the recognizer's exported recognition functions instead of the system.

This procedure may facilitate debugging the recognizer, but otherwise serves no purpose. It prevents use of other recognizers, including the system default recognizer. For a discussion of import libraries, refer to the section on the IMPLIB utility described in the *Environment and Tools* manual of the Microsoft VisualC++ documentation.

See Also

ConfigRecognizer, **ConfigHREC**, **UninstallRecognizer**

IsPenAware Overview

Overview

1.0 2.0

Checks the capability of an application to handle pen events by returning cached task pen flags.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions. Use **GetPenAppFlags** instead.

UINT IsPenAware( )

Return Value

Returns the registration flags word set by a previous call to the **SetPenAppFlags** function. If **SetPenAppFlags** has not been called, **IsPenAware** returns 0.

See Also

GetPenAppFlags, **SetPenAppFlags**

IsPenEvent Overview

Overview

1.0 2.0

Checks whether a mouse event was generated by the pen driver.

BOOL IsPenEvent(UINT *message*, **LONG** *lExtraInfo* **)**

Parameters

message

Windows mouse message being queried.

lExtraInfo

Value returned by **GetMessageExtraInfo** for the given message.

Return Value

Returns TRUE if the mouse event referenced by the *message* parameter was generated by the pen driver; otherwise, returns FALSE.

Comments

Mouse drivers that have not been updated to be compatible with pens may produce an event that cannot be distinguished from a real pen event. This has a very low probability of occurring.

KKConvert Overview

Overview

2.0

Activates the Kana-to-Kanji converter. (Japanese version only.)

BOOL KKConvert(HWND *hwndConvert*, **HWND** *hwndCaller*, **LPSTR** *lpBuf*, **UINT** *cbBuf*, **LPPOINT** *lpPoint* **)**

Parameters

hwndConvert

Handle to the window with the text to be converted.

hwndCaller

Handle to the window that calls **KKConvert**.

lpBuf

The text to be converted.

cBuf

The number of bytes (greater than 1) in *lpBuf*.

lpPoint

The positition where the Kana-to-Kanji converter will appear.

Return Value

Returns TRUE if the text is successfully converted; otherwise, returns FALSE.

Comments

If *lpBuf* is NULL, the currently-selected text in the window specified by *hwndConvert* will be used for conversion and then replaced with the converted text. If *lpBuf* is not NULL, the text in *lpBuf* will be converted and replaced with the converted text. If the length of the converted text is longer than *cbBuf*, the text will be truncated.

If the window referenced by *hwndConvert* is of the bedit class, *lpPnt* is ignored; otherwise, the center of the Kana-to-Kanji conversion is displayed at *lpPnt*.

If the window referenced by *hwndConvert* is of the hedit class and *lpPnt* is NULL, then the current caret position is used; otherwise, the client position (0,0) in the window referenced by *hwndConvert* is used.

MetricScalePenData Overview

Overview

1.0 2.0

Converts pen data points to one of the supported metric modes.

BOOL MetricScalePenData(HPENDATA *hpndt*, **UINT** *wPndtNew* **)**

Parameters

hpndt

Handle to a pen data object containing the points to be converted.

wPndtNew

Scaling metric to be used with the data, as listed here:

Constant	Description
PDTS_LOMETRIC	Each logical unit is mapped to 0.1 millimeter. Positive x is to the right; positive y is down.
PDTS_HIMETRIC	Each logical unit is mapped to 0.01 millimeter. Positive x is to the right; positive y is down.
PDTS_HIENGLISH	Each logical unit is mapped to 0.001 inch. Positive x is to the right; positive y is down. This is equivalent to PDTS_STANDARDSCALE.
PDTS_DISPLAY	This parameter scales the data, using **DPtoTP**. The pen data memory block is left in display coordinates.

Return Value

Returns TRUE if successful, or FALSE if *hpndt* is in a compressed state or if the data is not already in one of the metric modes such as PDTS_ARBITRARY.

Comments

The **MetricScalePenData** function converts pen coordinates between metric and English standard measurements. Metric units are 0.1 and 0.01 millimeter; English standard units are 0.001 inch. These scaling metrics form the same mapping mode set in the Windows function **SetMapMode**.

MetricScalePenData allows an application to transform pen data to the mapping mode set for a device context. This ensures that ink rendered in the device context appears in the proper scale.

Note the following caveats about **MetricScalePenData**:

- Because of rounding errors, scaling is not precisely reversible between mapping modes. Rounding errors can also adversely affect recognition accuracy if the data is later given to a recognizer. The problem arises when transforming the standard ink scale of HIENGLISH to a scale of lower resolution, a transformation that loses some of the original data. The lost data cannot be recovered, even if the coordinates are converted back into HIENGLISH.
- The scaling is not perfect and results in numerous "off-by-one" discrepancies, visible when displaying the scaled data.

Strictly speaking, the PDTS_DISPLAY scaling type is not a metric scale. To use it, the current scale of the data must be in PDTS_STANDARDSCALE units.

The effect of this call is similar to that of using the **TPtoDP** function on the array of points. A recognizer may not accurately recognize the resulting data. As with the other scales, the PDTS_DISPLAY is set in the **wPndts** member of the pen data header. If data is in PDTS_DISPLAY scale, **MetricScalePenData** cannot be called to scale it back to the other metric scales. No overflow checks are made. Because of rounding errors, scaling conversion is not perfectly reversible. Recognizers must recognize points that have been scaled to PDTS_STANDARDSCALE (equivalent to PDTS_HIENGLISH).

See Also

OffsetPenData, **ResizePenData**, PDTS_

OffsetPenData Overview

Overview

1.0 2.0

Offsets the coordinates in an **HPENDATA** memory block to make them relative to another origin.

BOOL OffsetPenData(HPENDATA *hpndt*, **int** *dx*, **int** *dy* **)**

Parameters

hpndt

Handle to a pen data object.

dx

Offset of x-axis; that is, the amount to move left or right. To move left, the *dx* value must be negative.

dy

Offset of y-axis; that is, the amount to move up or down. To move up, the *dy* value must be negative.

Return Value

Returns TRUE if successful, or FALSE if *hpndt* is in a compressed state.

Comments

For every point in *hpndt*, *dx* is added to the x-coordinate and *dy* is added to the y-coordinate. No overflow checks are made.

An application can use **OffsetPenData** to make points at display resolution relative to a particular window. If the window is then moved, the application need only call **OffsetPenData** again to move the data by the same amount, as shown in the example.

Example

The following sample code illustrates using the **OffsetPenData** function.

```
DWORDdwOrg;                                  // Store window origin
      .
      .
      .
// After creating window, note its current position
dwOrg = GetWindowOrg( hWnd );
      .
      .
      .
switch( wMsg )
{
      case WM_MOVE:
            dx    = (int) (LOWORD(lParam) - LOWORD(dwOrg)); // X increment
            dy    = (int) (HIWORD(lParam) - HIWORD(dwOrg)); // Y increment
            dwOrg = (DWORD) lParam;                                    //
Keep new org
            OffsetPenData( hpendata, dx, dy );
```

See Also

MetricScalePenData, **ResizePenData**

PeekPenInput Overview

Overview

2.0

Retrieves information about a specified pen packet in the pen input queue. For a definition of pen packet, see the description of **SetPenHook**.

int PeekPenInput(HPCM *hpcm*, **UINT** *idEvent*, **LPPOINT** *lppt*, **LPVOID** *lpvOem*, **UINT** *fuOemFlags* **)**

Parameters

hpcm

Handle to a pen collection. This is the return value from **StartPenInput**.

idEvent

The identifier of the packet to be retrieved. The *idEvent* is the low-order word of the value returned from the Windows **GetMessageExtraInfo** function when processing a WM_LBUTTONDOWN message.

lppt

Far pointer to a **POINT** structure. **PeekPenInput** copies the point corresponding to *idEvent* into the buffer pointed to by *lppt*.

lpvOem

The address of a buffer of OEM data in the packet. This parameter can be NULL if no OEM data is required.

fuOemFlags

Flags specifying which OEM data to retrieve. If this parameter is NULL, all of the OEM data provided by the tablet is returned in the order specified by the **rgoempeninfo** array in **PENINFO**.

These flags have an implicit order. For example, if pressure and barrel rotation are specified in that order, *cPntMax* pairs of these data are returned in the same order: [pressure, rotation], [pressure, rotation], and so on. (*cPntMax* is the number of **POINT** structures specified in **GetPenInput**.)

Constant	Description
PHW_PRESSURE	Retrieve pressure data.
PHW_HEIGHT	Retrieve height data.
PHW_ANGLEXY	Retrieve data pertaining to the x- and y-coordinates.
PHW_ANGLEZ	Retrieve data pertaining to the z-coordinates.
PHW_BARRELROTATION	Retrieve barrel-rotation data.
PHW_OEMSPECIFIC	Retrieve OEM-specific data.
PHW_PDK	Retrieve PDK_ data.

Return Value

Returns PCMR_OK if successful; otherwise, the return value can be one of the following:

Constant	Description
PCMR_INVALIDCOLLECTION	The *hpcm* handle is invalid

	because the calling application did not start input with **StartPenInput**.
PCMR_INVALID_PACKETID	*idEvent* is invalid.

Comments

Unlike **GetPenInput**, this function does not remove data from the pen input queue. It only returns information about the packet specified by *idEvent*.

Whereas *lppt* points into the pen input queue, *lpvOem* does not. If *lpvOem* is not NULL, it points to a buffer provided by the caller into which the OEM data are copied from the pen input queue.

The buffer that *lpvOem* points to must be large enough to hold the requested OEM data copied from the packet. The size of each packet is the width specified in the **cbOemData** member of the **PENINFO** structure, plus sizeof(UINT) if PDK_ values are required.

See Also

GetPenInput, **PENPACKET**, PDK_

PenDataFromBuffer Overview

Overview

2.0

Creates a **HPENDATA** object from serial data in a buffer. The buffer must have been previously written by the **PenDataToBuffer** function.

LONG PenDataFromBuffer(LPHPENDATA *lphpndt*, **UINT** *gmemFlags*, **LPBYTE** *lpBuffer*, **LONG** *cbBuf*, **LPDWORD** *lpdwState* **)**

Parameters

lphpndt

Pointer to an uninitialized **HPENDATA** handle. If **PenDataFromBuffer** returns successfully, *lphpndt* points to a new **HPENDATA** object containing a copy of the serial points.

gmemFlags

Flag that specifies whether or not the Windows **GlobalAlloc** function should create a shared memory object when the pen data object is created. This should be either 0 or GMEM_DDESHARE. The GMEM_MOVEABLE and GMEM_ZEROINIT flags are added to this value and other GMEM_ flags are ignored.

lpBuffer

Pointer to a byte buffer containing serial data.

cbBuf

Size of the buffer, which must be at least 64 bytes in size. If the buffer serves as an intermediate holding area, it need not be as large as the final **HPENDATA** object. To create the object, the application must call **PenDataFromBuffer** successively, each time reading a new section of data into the buffer that *lpBuffer* points to before the call. The example below illustrates this technique by filling an **HPENDATA** object in stages, reading data from a file in *cbBuf* increments.

lpdwState

Address of a DWORD variable used by the system to maintain the transfer state. The DWORD variable must be initialized to 0 before the first call to **PenDataFromBuffer**. Between successive calls to **PenDataFromBuffer**, the application must not alter the value that *lpdwState* points to. *lpdwState* can be NULL to signify that the buffer contains the entire data set for the **HPENDATA** object. This implies that subsequent calls to **PenDataFromBuffer** are not necessary.

Return Value

If successful, **PenDataFromBuffer** returns the number of bytes transferred from the buffer. If the size of the pen data is larger than the buffer, the return value is equal to the buffer size passed in *cbBuf*. A value of 0 indicates no more data to transfer. If there is an error, one of the following negative values is returned:

Constant	Description
PDR_ERROR	Parameter or overflow error.
PDR_MEMERR	Memory error.

Comments

The data being provided by the application must have been previously written by the **PenDataToBuffer** function. The application cannot modify this data in any way. Embedded values within the first 64 bytes

provide information to **PenDataFromBuffer** about the size of the pen data.

PenDataFromBuffer creates an **HPENDATA** object and provides a handle to it. The application must destroy the object when finished. The *lphpndt* argument points to a valid **HPENDATA** handle only if the function returns PDR_OK.

While this function is reconstituting the **HPENDATA** object, the application must not attempt to use it in any way because it will be invalid until the last buffer is read.

Example

The following example shows how to create a **HPENDATA** object from a file (*hfile*) that has already been opened for reading. Before reading the pen data, its length is retrieved from the file:

```
#define cbBufMax 1024

HPENDATA NEAR PASCAL ReadPenData( HFILE hfile )
{
      HPENDATA          hpndt = NULL;
      LONG              cb, cbRead, cbHpndt;
      BYTE              lpbBuf[cbBufMax];       // Buffer
      DWORD       dwState = 0L;                 // Must initialize to 0

      if (!hfile
            || (cb = _lread(hfile, &cbHpndt, sizeof(DWORD))) == HFILE_ERROR
            || cb != sizeof(LONG))
                  return NULL;

      while (cbHpndt > 0)
      {
            if ((cbRead = _lread( hfile, lpbBuf, min(cbHpndt, cbBufMax )))
                  == HFILE_ERROR
                  || PenDataFromBuffer( &hpndt, 0, lpbBuf,
                                          cbBufMax, &dwState ) < 0)
            {
                  if (hpndt)
                        DestroyPenData( hpndt );
                  return NULL;
            }
            cbHpndt -= cbRead;
      }

      return hpndt;
}
```

See Also

PenDataToBuffer, GetPenDataAttributes

PenDataToBuffer Overview

Overview

2.0

Writes the data in an existing **HPENDATA** object to a serial buffer.

LONG PenDataToBuffer(HPENDATA *hpndt*, **LPBYTE** *lpBuffer*, **LONG** *cbBuf*, **LPDWORD** *lpdwState* **)**

Parameters

hpndt

Handle to the **HPENDATA** object.

lpBuffer

Pointer to an empty buffer.

cbBuf

Size of the buffer in bytes. The buffer must be at least 64 bytes in size. If the buffer serves as an intermediate holding area, it need not be as large as the **HPENDATA** object. To read all data from the object in this case, the application must call **PenDataToBuffer** successively, each time copying the data from the buffer that *lpBuffer* points to before the next call. The example below illustrates this technique by writing an **HPENDATA** object in *cbBuf* increments to a file.

lpdwState

Address of a DWORD variable used by the system to maintain the transfer state. The DWORD variable must be initialized to 0 before the first call to **PenDataToBuffer**. Between successive calls to **PenDataToBuffer**, the application must not alter the value that *lpdwState* points to. *lpdwState* can be NULL to signify that the buffer is large enough to contain the entire **HPENDATA** object. This implies that subsequent calls to **PenDataToBuffer** are not necessary.

Return Value

If successful, **PenDataToBuffer** returns the number of bytes transferred into the buffer. If the size of the pen data is larger than the buffer, the return value is equal to the buffer size passed in *cbBuf* until the final transfer, when it is typically some smaller value. A value of 0 indicates no more data to transfer. If there is an error, one of the following negative values is returned:

Constant	Description
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Memory error.
PDR_PNDTERR	Invalid **HPENDATA** object.
PDR_VERSIONERR	Could not convert old **HPENDATA** object.

Comments

The buffer need not be large enough to accommodate the entire **HPENDATA** object. To allocate a buffer large enough for a single transfer, the application can determine the required size with the **GetPenDataAttributes** subfunction GPA_SIZE .

Example

The following example shows how to save an **HPENDATA** object to a file (hfile) that has already been opened for writing. The length of the pen data is saved in the file before writing the pen data itself:

```
#define cbBufMax 1024
```

```
BOOL NEAR PASCAL WritePenData( HFILE hfile, HPENDATA hpndt )
{
     BYTE    lpbBuf[cbBufMax];
     DWORD   dwState = 0L;                    // Must initialize to zero
     LONG    cb;
     LONG    cbSize;

     if (!hfile || !hpndt)
           return FALSE;

     // Get size and save to file
     if (GetPenDataAttributes(hpndt, (LPVOID)&cbSize, GPA_SIZE) < 0)
           return FALSE;

     // write size of pen data to file so that it can be used while reading
it back
     if (_lwrite(hfile, &cbSize, sizeof(LONG)) == HFILE_ERROR)
           return FALSE;

     // Write the pen data in chunks, and repeat until done
     while ((cb = PenDataToBuffer(hpndt, lpbBuf,
                                          cbBufMax, &dwState )) > 0L)
     {
           if (_lwrite( hfile, lpbBuf, (UINT)cb ) == HFILE_ERROR)
                 return FALSE;
     }

     return (cb >= 0L);                  // Return TRUE if cb >= 0
}
```

See Also

PenDataFromBuffer

PostVirtualKeyEvent Overview

Overview

1.0 2.0

Posts a virtual key-code event to Windows.

void PostVirtualKeyEvent(UINT *vk*, **BOOL** *fUp* **)**

Parameters

vk

Virtual key. This argument takes a Windows VK_ constant as defined in the WINDOWS.H include file. Depending on the key, this is either the key scan code or the ASCII equivalent to represent a character key. For example, VK_A has the value "A."

fUp

Key-transition flag. This parameter should be FALSE to specify that the key is down or TRUE to specify that it is up.

Return Value

This function does not return a value.

Comments

PostVirtualKeyEvent does not check the virtual key code for errors.

Normally, an application should follow a key-down message with a corresponding key-up message to accurately simulate the actual events from the keyboard. You can post repeating keys by calling **PostVirtualKeyEvent** consecutively, one call per repeat, with *fUp* set to FALSE. End the sequence with a single call to **PostVirtualKeyEvent** with *fUp* set to TRUE.

The events are posted to the system message queue. The application with the input focus can receive the messages by calling the Windows **GetMessage** or **PeekMessage** function.

See Also

AtomicVirtualEvent, **PostVirtualMouseEvent**

PostVirtualMouseEvent Overview

Overview

1.0 2.0

Posts a virtual mouse code to Windows.

void PostVirtualMouseEvent(UINT *wMouseFlags*, **int** *xPos*, **int** *yPos* **)**

Parameters

wMouseFlags

Flags indicating the type of mouse event. This can be one or more of the following values, combined by a bitwise-OR operator.

Constant	Description
VWM_MOUSEMOVE	Simulates a change in the mouse cursor position. This flag can be combined with any of the other flags in this table.
VWM_MOUSELEFTDOWN	Simulates pushing the left mouse button.
VWM_MOUSELEFTUP	Simulates releasing the left mouse button.
VWM_MOUSERIGHTDOWN	Simulates pushing the right mouse button.
VWM_MOUSERIGHTUP	Simulates releasing the right mouse button.

xPos

The x-axis position in screen coordinates.

yPos

The y-axis position in screen coordinates.

Return Value

This function does not return a value.

Comments

The x- and y-axis positions are absolute positions in screen coordinates. Note that the x and y values should not exceed the screen-resolution limits. Values greater than the maximum resolution in the x-direction (640 for standard VGA) or the y-direction (480 for standard VGA) cause an overflow.

The events are posted to the system message queue. The application with the input focus can receive the messages by calling the Windows **GetMessage** or **PeekMessage** message.

Because of the way Windows interprets mouse messages, the calling application must be careful about the order in which events are sent to the system. A message that represents both a button-state transition and a move generates first a Windows event for the button transition at the current pointer location and then a move to the new location. To simulate a move to a new location followed by a button transition, the application must make separate calls to **PostVirtualMouseEvent** for each simulated event.

Example

When posting events, the caller should bracket the calls by calls to **AtomicVirtualEvent**, which locks out pen packets while the application is posting simulated mouse events. For example, the following code fragment posts a mouse event:

```
AtomicVirtualEvent(TRUE);
//
//  ... PostVirtualMouseEvent calls go here
//
AtomicVirtualEvent(FALSE);
```

The Windows **GetMessageExtraInfo** function returns 0 for any messages generated by **PostVirtualMouseEvent**.

See Also

AtomicVirtualEvent, **PostVirtualKeyEvent**

ProcessHRC Overview

Overview

2.0

Gives a recognizer sufficient time for intermediate processing of pen input. A recognizer must export this function.

int ProcessHRC(HRC *hrc*, **DWORD** *dwTimeMax* **)**

Parameters

hrc

Handle to the **HRC** object for the recognizer.

dwTimeMax

The maximum time in milliseconds that the recognizer should process before returning from this call. This parameter can also be one of the following time-out codes:

Constant	Description
PH_MIN	The recognizer should take only a very small amount of time to process the input, typically 50 milliseconds.
PH_DEFAULT	The recognizer should take a moderate amount of time to process the input, typically 200 milliseconds.
PH_MAX	The recognizer should take as much time as required to complete recognition.

Return Value

If there are no errors, returns one of the following values:

Constant	Description
HRCR_OK	Processing is successful.
HRCR_INCOMPLETE	The recognizer is still processing the current batch of input.
HRCR_GESTURE	The recognizer has recognized a possible gesture. This can be returned before the recognition process is complete. If the processing completes, HRCR_COMPLETE is always returned, even for gestures.
HRCR_COMPLETE	The recognizer completed processing and does not expect any more input.

To indicate an error, **ProcessHRC** returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.

Comments

ProcessHRC returns if the time specified by *dwTimeMax* elapses before recognition is complete.

In an operating environment that does not use threads, this function allows an application to provide some time for the recognizer to process input. By checking the return value, the application is able to monitor the progress, specifically whether a gesture is a recognition candidate. If the function returns HRCR_GESTURE, the application can call **GetResultsHRC** to determine whether the gesture should be acted on immediately.

Typically, the return value is HRCR_OK if the current batch of ink input has been processed by the recognizer, and HRCR_INCOMPLETE if the recognizer has not yet finished processing.

If **ProcessHRC** is called with PH_MAX, recognition is complete only if **EndPenInputHRC** has been called to notify the recognizer that no more results are expected. The return value in this case is HRCR_COMPLETE, and the application is free to get and display final results. If the application supplies further input at this point, it has the effect of canceling the **EndPenInputHRC** call, although this procedure is not recommended for reasons of efficiency.

However, PH_MAX may result in poorer performance, since further processing in the system is blocked until **ProcessHRC** returns. Instead, the application can call **ProcessHRC** in an idle loop or a separate thread, calling it repeatedly with smaller time allotments until the function returns HRCR_COMPLETE. Note that if a separate thread is used to finish processing, the main thread can call this function with PH_MIN from time to time to determine if processing has finished (that is, checking for the HRCR_COMPLETE return value).

The first time that **ProcessHRC** is called for a particular recognition context, functions that set its state cannot be used for the remainder of that context's existence. The following functions return an error if they are called before **ProcessHRC** returns HRCR_COMPLETE:

- **EnableGestureSetHRC**
- **EnableSystemDictionaryHRC**
- **SetAlphabetHRC**
- **SetAlphabetPriorityHRC**
- **SetBoxAlphabetHRC**
- **SetWordlistCoercionHRC**
- **SetGuideHRC**
- **SetInternationalHRC**
- **SetMaxResultsHRC**
- **SetWordlistHRC**

See Also

EndPenInputHRC, **GetResultsHRC**

ProcessWriting Overview

Overview

1.0 2.0

Processes handwriting.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions. Use **DoDefaultPenInput** instead.

REC ProcessWriting(HWND *hwnd*, **LPRC** *lprc* **)**

Parameters

hwnd

Window to receive messages. This parameter must not be NULL.

lprc

Address of **RC** structure to use for recognition. This parameter can be NULL.

Return Value

Returns values less than 0 if the application should treat the event as a mouse event instead of a pen event. Return values less than 0 occur if the event did not come from a pen, the user performed a press-and-hold action (REC_POINTEREVENT), or an error occurred—for example, running out of memory.

Comments

The **ProcessWriting** function is similar to **Recognize** except that **ProcessWriting** also takes care of inking, removing the ink, and converting the results message to standard Windows messages.

Depending on the existing code in an application, **ProcessWriting** may not be suitable for making an application pen-aware. This function can also limit the power of a pen interface.

If *lprc* is NULL, a default **RC** structure is created for the application. The default **RC** structure contains all system defaults and the inking is constrained to the client area of *hwnd*. If *lprc* points to a valid **RC** structure, the **rectBound** member of the **RC** structure is used to constrain the inking. Regardless of whether the application provides an **RC** or not, **ProcessWriting** assumes a value of RRM_COMPLETE for the **wResultMode** member. See **RC** for a description of **wResultMode** and the RRM_ values.

After the writing is completed, the ink is removed before any messages are sent to *hwnd*. After the ink is removed, the screen is updated and *hwnd* receives a WM_RCRESULT message. If the application processes this message, it should return TRUE. In this case, no further messages are sent.

If the application returns FALSE, **ProcessWriting** performs the default conversion of the results message to standard Windows messages, as shown in the following table. The messages are sent rather than posted. Note that the **DefWindowProc** function returns 0 when processing the WM_RCRESULT message.

Results message	Windows message
SYV_BACKSPACE	WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hot spot of the gesture, followed by WM_CHAR specifying a backspace.
SYV_CLEAR	WM_CLEAR.

SYV_CLEARWORD	WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_LBUTTONUP at the same point, followed by WM_CLEAR.
SYV_COPY	WM_COPY.
Results message	**Windows message**
SYV_CORRECT	WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_LBUTTONUP at the hot spot of the gesture, followed by WM_COPY. At this point the Edit Text dialog box is activated; it retrieves text from the Clipboard. This uses the existing selection, if any. The previous contents of the Clipboard are lost.
SYV_CUT	WM_CUT.
SYV_EXTENDSELECT	WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hot spot of the gesture. The MK_SHIFT flag is set for the *wParam* of these messages.
SYV_LASSO	WM_LBUTTONDOWN at upper-left corner of selected area, followed by WM_MOUSEMOVE message, followed by WM_LBUTTONUP at the lower-right corner of selected area.
SYV_PASTE	WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hot spot of the gesture, followed by WM_PASTE.
SYV_RETURN	WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hot spot of the gesture, followed by WM_CHAR specifying a carriage return.
SYV_SPACE	WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hot spot of the gesture, followed by WM_CHAR specifying a space.
SYV_TAB	WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hot spot of the gesture, followed by WM_CHAR specifying a tab.
SYV_UNDO	WM_UNDO.
text	One WM_CHAR message per character of text.

The SYV_ symbol values in the previous table identify gestures. To see a complete list of symbol values, refer to Chapter 13, "Pen Application Programming Interface Constants."

The *lParam* of a WM_RCRESULT message generated by **ProcessWriting** is a far pointer to an **RCRESULT** structure. By default, when an application receives a WM_RCRESULT message, the **hpendata** member of the **RCRESULT** structure is NULL. If you need the **HPENDATA** handle, set the RCO_SAVEHPENDATA flag in the **lRcOptions** member of the **RC** structure. In this case, the calling application is responsible for destroying the **HPENDATA** object.

See Also

DoDefaultPenInput, **InitRC**, **Recognize**, REC_, SYV_, RCO_

ReadHWL Overview

Overview

2.0

Reads a word list from a file.

int ReadHWL(HWL *hwl*, **HFILE** *hfile* **)**

Parameters

hwl

A handle to an empty word list.

hfile

A handle to a file previously opened for reading.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or file or other error.
HRCR_MEMERR	Insufficient memory.

Comments

The words are expected as ANSI text, one word per line, followed by a carriage return and linefeed. In this context, a word can represent a phrase and contain spaces or other noncharacters, such as "New York" and "ne'er-do-well." Empty lines or lines containing only spaces or tabs are allowed but ignored.

The file that *hfile* refers to must already exist and be open for reading.

The *hwl* parameter must be the handle of an empty word list created by **CreateHWL**. If the word list is not empty, **ReadHWL** returns HRCR_ERROR.

Once the file is read, it can be closed immediately.

Example

The following example demonstrates how to provide a word list to constrain recognition results to the words contained in the fictitious file COUNTRY.LST:

```
HWL    hwlCountries = CreateHWL( NULL, NULL, WLT_EMPTY, 0L );
OFSTRUCT     ofStruct;
HFILE  hfile = OpenFile( "country.lst", &ofStruct, OF_READ );

if (hfile != HFILE_ERROR)
{
ReadHWL(hwlCountries, hfile);
_lclose(hfile);
}
else
ErrorMsg(FILEOPEN);
.
.
.
```

```
if (hrc = CreateCompatibleHRC( NULL, NULL ))
{
SetWordlistHRC( hrc, hwlCountries );          // Set list into HRC
SetWordlistCoercionHRC( hrc, SCH_FORCE );  // Force match
.
.     // Code that collects and recognizes input here
.
}
```

See Also

CreateHWL, WriteHWL

Recognize Overview

Overview

1.0 2.0

Begins sampling pen data and converts tablet input to recognized symbols.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

REC Recognize(LPRC *lprc* **)**

Parameters

lprc

Address of an **RC** structure.

Return Value

Returns an REC_ value. See the "Comments" section for a description of the return values.7

Comments

The **RC** structure that *lprc* points to contains the parameters that control recognition. The system sends recognition results via the WM_RCRESULT message to the window indicated by the **hwnd** member of the **RC**. All results messages are sent before **Recognize** returns. Multiple result messages can be sent if the application asks for results to be sent to the application before all input has been completed (as indicated by the **wResultMode** member of the **RC** structure).

An application that uses version 1.0 recognizers should call **Recognize** when the input session begins, signaled by the WM_LBUTTONDOWN message.

The value REC_OK is used in the *wParam* of the WM_RCRESULT message to indicate that more data is coming. Return values of greater than 0 signal normal successful completion. Return values of less than 0 indicate abnormal termination. Return values of less than REC_DEBUG are reserved for return values from

debugging versions of the system or recognizer. If an application creates a condition that would be caught in a debugging version while running a nondebugging version, the results are undefined.

Each return value can be the *wParam* value of the WM_RCRESULT message or the return value for **Recognize**. The *wParam* value of the last WM_RCRESULT message generated by a call to **Recognize** is the return value of **Recognize**. Some error conditions, such as REC_OOM or REC_NOTABLET, are returned without generating any corresponding WM_RCRESULT message.

All of the values listed in the following table are in the debug version only. No WM_RCRESULT message is generated if these values are returned by **Recognize**.

Constant	Description
REC_ALC	Invalid enabled alphabet.
REC_BADEVENTREF	Returned when the **wEventRef** member in the *lprc* structure is invalid.
REC_CLVERIFY	Invalid verification level.
REC_DEBUG	All debugging return values are less than

	this.
REC_DICT	Invalid dictionary parameters.
REC_ERRORLEVEL	Invalid error level.
REC_GUIDE	Invalid **GUIDE** structure.
REC_HREC	Invalid recognition handle.
REC_HWND	Invalid handle to window to send results to.
REC_INVALIDREF	Invalid data reference parameter.
REC_LANGUAGE	Returned by the recognizer when the **lpLanguage** member contains a language that is not supported by the recognizer. Call **ConfigRecognizer** with the WCR_QUERYLANGUAGE subfunction to determine whether a particular language is supported.
REC_NOCOLLECTION	In version 1.0, was returned by **GetPenHwData** if collection mode has not been set. Not used now.
REC_OEM	Error codes less than or equal to REC_OEM are specific to the recognizer.
REC_PCM	Invalid **lPcm** member in the **RC** structure. There is no way for the recognition to end.
REC_RECTBOUND	Invalid rectangle.
REC_RECTEXCLUDE	Invalid rectangle.
REC_RESULTSMODE	Unsupported results mode requested.

See Also

InitRC, **RecognizeData**, **GetPenHwEventData**, **RC**, REC_

RecognizeData Overview

Overview

1.0 2.0

Converts the data in an **HPENDATA** object to recognized symbols.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

REC RecognizeData(LPRC *lprc*, **HPENDATA** *hpndt* **)**

Parameters

lprc

Address of an **RC** structure.

hpndt

Handle to an **HPENDATA** object.

Return Value

Returns REC_DONE if successful, or an REC_ error code if an error occurs.

Comments

RecognizeData recognizes data in an **HPENDATA** object and returns the results to the window specified in the **RC** structure. **RecognizeData** is similar to **Recognize**. The difference is that in **RecognizeData**, the input data comes from a buffer of points already collected instead of from the tablet driver. Members pertaining to the end of recognition in the **RC** structure are ignored.

RecognizeData can return REC_BUSY if the recognizer is not reentrant. A recognizer is not guaranteed to return the same results for identical input. This is because persistent states, such as the current average size of writing or the position of the baseline, can affect recognition results. In addition, training may change the prototypes against which the data is being compared.

RecognizeData attempts to convert the pen data to PDTS_STANDARDSCALE if it is not already in standard scale. If the conversion fails (for example, because the data was in an application-specific scale PDTS_ARBITRARY), the data is still passed to the recognizer. A recognizer may return an error code (REC_BADHPENDATA) for data in a scale it cannot handle.

See Also

InitRC, **Recognize**, **GetPenHwEventData**, **RC**, REC_, PDTS_

RedisplayPenData Overview

Overview

1.0 2.0

Redraws the pen data in the same manner as originally inked.

BOOL RedisplayPenData(HDC *hdc*, **HPENDATA** *hpndt*, **LPPOINT** *lpDelta*, **LPPOINT** *lpExt*, **int** *nInkWidth*, **DWORD** *rgbColor* **)**

Parameters

hdc

Handle to a device context. The mapping mode should be MM_TEXT.

hpndt

Handle to a pen data object. The pen data must be scaled to PDTS_DISPLAY or PDTS_STANDARDSCALE.

lpDelta

An offset, in logical units, that is subtracted from the pen data points to position the ink. If *lpDelta* is NULL, there is no offset.

lpExt

Extent, in logical units, for scaling. If *lpExt* is NULL, no scaling is performed.

nInkWidth

Width of the ink to be drawn, in pixels (1 to 15). If *nInkWidth* is -1, the strokes are rendered using the original ink width stored in the stroke header. An ink width of 0 causes the function to simply return TRUE.

rgbColor

RGB value of the color to draw the ink. If *rgbColor* is 0xFFFFFFFF, the strokes are rendered using the original ink color stored in the stroke header.

Return Value

Returns TRUE if successful; otherwise FALSE.

Comments

The *nInkWidth* and *rgbColor* values override the pen currently selected for the *hdc* device context.

If the mapping mode of the *hdc* device context is not MM_TEXT, two problems can occur:

- **RedisplayPenData** uses **TPtoDP** to prepare the pen data points for rendering. After this, the points are in MM_TEXT coordinates; this assumes an MM_TEXT device context for display. If the device context is in a different mapping mode, the ink coordinates will not be correct. Even if you use the ink-scaling functions to bypass this problem, you will still encounter rounding-error problems between the two scalings.
- No matter what scaling is done, rounding errors occur when converting between modes. These errors cause the ink to shift slightly when repainted.

For any rendering into a device context that represents anything other than a display device context, **DrawPenDataEx** should be used. This is because **RedisplayPenData** makes assumptions that are not optimal for other devices such as printers or metafiles.

RedisplayPenData provides the ability to re-create original inking perfectly. To do this, an application can use either of two methods:

- After the input session ends and data is collected into an **HPENDATA** object, store the current origin of the window containing the ink. When calling **RedisplayPenData** to redraw the ink, supply the origin value in the *lpDelta* argument, set *lpExt* to NULL, and set the mapping mode of the device context to MM_TEXT. Only ink data with a common window origin can be merged into a single **HPENDATA**.
- In the second method, the application must call two Pen functions immediately after collecting the data into an **HPENDATA** object. The first call to **MetricScalePenData** converts the pen data to display coordinates. The second call to **OffsetPenData** sets the display coordinates relative to the window containing the original ink. To display, the application must call **RedisplayPenData** with *lpDelta* and *lpExt* set to NULL and the mapping mode of the device context set to MM_TEXT. If the application adopts this method for multiple **HPENDATA** objects, it can later merge them to form a single **HPENDATA** object (up to the 64K limit).

The second method has the advantages of simplicity and data compression. See the description of **MetricScalePenData** for a discussion of the limitations of converting data to display resolution.

Since the pen data has the origin of (0,0) based on the upper-left corner of the display, applications must move from a screen-relative position to a position relative to the device context. To do this, subtract the origin of the device context (in screen coordinates) from the object currently residing in screen-coordinate space.

The *lpDelta* parameter enables the application to render ink relative to the window instead of relative to the screen. An application should call the **ClientToScreen** function for (0,0) to find the proper screen coordinates to be placed in the **lpDelta* **POINT** structure. Once this is done, the pen data is rendered at the appropriate location in window coordinates. If *lpDelta* is NULL, no offset for the data is assumed.

The *lpExt* argument specifies the extents into which the data should be scaled. If extents are provided, data is scaled into a rectangle described by *lpDelta* and *lpExt*. The values of x and y in *lpExt* and *lpDelta* are in the mapping mode of the device context into which the data is rendered.

RedisplayPenData displays pen data with a square graphical device interface (GDI) pen brush for maximum drawing speed. When displaying wide lines of ink, this optimization can cause the ends of abutting lines to appear blocky. If you prefer a smoother look to the joints of wide lines at the expense of rendering speed, draw the ink with **DrawPenData**, **DrawPenDataEx**, or **DrawPenDataFmt** instead of **RedisplayPenData**. These functions draw wide lines by flood-filling a region, thus smoothing the ends.

See Also

DrawPenData, **DrawPenDataEx**, PDTS_

RegisterPenApp Overview

Overview

1.0 2.0

Notifies the pen system that the application edit controls should be replaced with hedit controls. This function is required only for applications that specify EDIT class (instead of HEDIT class) for control windows with versions of Windows earlier than Windows 95.

Note that this function has been superseded by the **SetPenAppFlags** function in the 2.0 version of the Pen API, although calling **RegisterPenApp** is still supported. See **SetPenAppFlags** for more information.

void RegisterPenApp(UINT *fuFlags*, **UINT** *uVersion* **)**

RemovePenDataStrokes Overview

Overview

2.0

Removes strokes from an **HPENDATA** object.

int RemovePenDataStrokes(HPENDATA *hpndt*, **UINT** *iStrk*, **UINT** *cStrks* **)**

Parameters

hpndt

Handle to the **HPENDATA** object.

iStrk

Zero-based index of the first stroke to remove. This value can be IX_END to remove the last stroke. The function fails if *iStrk* is greater than the number of strokes in the pen data object.

cStrks

Count of strokes to remove. If this value is greater than the number of strokes after the specified stroke index, the stroke indexed by *iStrk* and all following strokes are removed. *cStrks* can be IX_END to remove all strokes from *iStrk* onward.

Return Value

Returns PDR_OK if successful; otherwise, the return value can be one of the following negative values:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Out of memory.
PDR_PNDTERR	Invalid pen data object.
PDR_STRKINDEXERR	Invalid stroke index.
PDR_VERSIONERR	Could not convert old pen data object.

Comments

RemovePenDataStrokes removes the number of strokes specified by *cStrks*, starting at the stroke specified by *iStrk*. Use **ExtractPenDataPoints** to remove points from a particular stroke of the pen data object.

See Also

ExtractPenDataPoints, **InsertPenDataPoints**, **InsertPenDataStroke**

ResizePenData Overview

Overview

1.0 2.0

Scales ink in an **HPENDATA** object into an arbitrarily sized rectangle.

BOOL ResizePenData(HPENDATA *hpndt*, **LPRECT** *lprect* **)**

Parameters

hpndt

Handle to a pen data object.

lprect

Address of a bounding rectangle, or NULL.

Return Value

Returns TRUE if successful; otherwise, the return value is FALSE.

Comments

This function changes the physical size of the object without changing the meaning of the measurements. Use the **MetricScalePenData** function to convert the data to one of the supported metric modes of measurement.

ResizePenData physically resizes the data in *hpndt* to the bounding rectangle dimensions given by the *lprect* parameter. Data from *hpndt* is mapped to the new rectangle. If *lprect* is NULL, this function recalculates the bounding rectangle (the **rectBound** member in the **PENDATAHEADER** structure). For example, consider the case of pen data with PDTS_HIMETRIC scaling bounded by the square (500, 600, 1500, 1600). To double the size, set *lprect* to (500, 600, 2500, 2600).

See Also

OffsetPenData, **MetricScalePenData**, PDTS_

ResultsHookHREC

2.0

The **ResultsHookHREC** function is an application-defined callback function that provides the application with the opportunity to view all recognition results before they are returned to the application. The name **ResultsHookHREC** is a placeholder; the function can have any name.

BOOL CALLBACK ResultsHookHREC(HREC *hrec*, **HRC** *hrc*, **WORD** *wHooktype*, **UINT** *cResults*, **UINT** *cAlt*, **LPVOID** *rgresults* **)**

Parameters

hrec

Module handle of the recognizer library whose results are being hooked.

hrc

Handle to the **HRC** object for the recognizer that *hrec* refers to.

wHooktype

Type of hook. This can be one of the following values:

RHH_STD

Standard results generated by **GetResultsHRC**.

RHH_BOX

Boxed results generated by **GetBoxResultsHRC**.

cResults

Count of results available.

cAlt

Count of box alternatives. This is valid only if *wHooktype* is RHH_BOX.

rgresults

An array of result objects. The object type depends on *wHooktype*. If RHH_STD, *rgresults* should be cast as **LPHRCRESULT** and the array receives *cResults* **HRCRESULT** objects. If RHH_BOX, *rgresults* should be cast as **LPBOXRESULTS** and the array receives *cResults* **BOXRESULTS** structures.

Return Value

The application hook function should return TRUE to indicate that it has processed the data and that the recognizer should do no further processing. In this case, it is the application's responsibility to destroy the results and inksets, if any; otherwise, the hook function should return FALSE.

See Also

SetResultsHookHREC

SetAlphabetHRC Overview

Overview

2.0

Specifies which alphabet should be used in an **HRC** object.

int SetAlphabetHRC(HRC *hrc*, **ALC** *alc*, **LPBYTE** *rgbfAlc* **)**

Parameters

hrc

Handle to the **HRC** object.

alc

Alphabet. This value is one or more ALC_ values combined using the bitwise-OR operator.

rgbfAlc

Array of bits if *alc* contains ALC_USEBITMAP; otherwise, it can be NULL.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

Some recognizers may not support ALC_ values and alphabet priorities. An application should check for HRCR_UNSUPPORTED when using this function.

The following values may require Japanese, wide-character, or recognizer-specific support: ALC_DBCS, ALC_JIS1, ALC_KANJI, ALC_OEM, ALC_HIRAGANA, and ALC_KATAKANA. In addition, ALC_RESERVED is reserved for future use and is ignored. Recognizers, such as the Microsoft Handwriting Recognizer (GRECO.DLL) for default American English, can return HRCR_OK even if some of these values are set.

The size of the *rgbfAlc* array, if used, must be large enough to accommodate 256 bits (32 bytes). If the *n*th bit is set, then the *n*th ANSI character is recognizable. Bits representing characters less than 32 (space) currently have no meaning.

The ALC_GESTURE value is ignored, even if it is part of the *alc* parameter. See **EnableGestureSetHRC**.

For a description of alphabets and their relationship to a recognizer, see "Configuring the HRC" in Chapter 5, "The Recognition Process." For a list of alphabet codes, see Chapter 13, "Pen API Constants."

See Also

EnableGestureSetHRC, **GetAlphabetHRC**

SetAlphabetPriorityHRC, ALC_

SetAlphabetPriorityHRC Overview

Overview

2.0

Specifies the priority of alphabet sets in an **HRC** object.

int SetAlphabetPriorityHRC(HRC *hrc*, **ALC** *alc*, **LPBYTE** *rgbfalc* **)**

Parameters

hrc

Handle to the **HRC** object.

alc

Alphabet priority. This value is one or more ALC_ values combined using the bitwise-OR operator.

rgbfalc

Address of a 256-bit (32-byte) buffer whose bits map to ANSI single-byte characters if *alc* contains ALC_USEBITMAP; otherwise, it can be NULL.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

Some recognizers may not support ALC_ values and priorities. An application should check for HRCR_UNSUPPORTED when using this function.

For a description of how a recognizer uses alphabet priority, see "Configuring the HRC" in Chapter 5, "The Recognition Process." For a list of alphabet codes, see Chapter 13, "Pen Application Programming Interface Constants."

See Also

GetAlphabetHRC, **SetAlphabetHRC**, **GetAlphabetPriorityHRC**, ALC_

SetBoxAlphabetHRC Overview

Overview

2.0

Specifies the alphabet codes to use for a range of boxes.

int SetBoxAlphabetHRC(HRC *hrc*, **LPALC** *rgalc*, **UINT** *cAlc* **)**

Parameters

hrc

Handle to an **HRC** object.

rgalc

An array of *cAlc* ALC_ values. The array is mapped onto boxes starting at box zero.

cAlc

Number of ALC_ values in *rgalc*. This should match the number of boxes. If this parameter is 0, **SetBoxAlphabetHRC** simply returns 0.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

SetBoxAlphabetHRC applies only when an **HRC** has been configured for box guides with the SetGuideHRC function. Although SetAlphabetHRC can also specify an alphabet set for boxed input, it attaches the same alphabet setting to all boxes indiscriminately. **SetBoxAlphabetHRC** offers greater control by allowing an application to set different alphabets for individual boxes of a single **HRC**.

Example

For example, consider a boxed entry on a requisition form that expects a part number consisting of five characters. The first two characters are uppercase letters, the next two are numerals, and the last character can be either another numeral or a lowercase revision code. The following example demonstrates how to configure the **HRC** for this hypothetical scenario:

```
#define  PART_LEN    5                  // Five characters in entry

HRC      hrcPart;                       // HRC for parts entry
GUIDE    guidePart;                     // GUIDE for parts entry
ALC      alcPart[PART_LEN];             // Array of ALC_ codes for entry
     .
     .                                   // Initialize the GUIDE here
     .
guidePart.cHorzBox = PART_LEN;          // Number of boxes in entry
guidePart.cVertBox = 1;                 // Single row
guidePart.cyMid    = 0;                 // No midline
```

```
iret = SetGuideHRC( hrcPart, (LPGUIDE)&guidePart, 0 );

if (iret == HRCR_OK)
{
     alcPart[0] = alcPart[1] = ALC_UCALPHA;  // Uppercase in boxes 1-2
     alcPart[2] = alcPart[3] = ALC_NUMERIC;  // Numerals in boxes 3-4
     alcPart[4] = ALC_LCALPHA | ALC_NUMERIC; // Lower or numeral in box 5

     // Map alphabet codes onto boxes of parts number entry
     SetBoxAlphabet( hrcPart, (LPALC)&alcPart, PART_LEN );
}
```

See Also

GetAlphabetHRC, SetAlphabetHRC, ALC_

SetGlobalRC Overview

Overview

1.0 2.0

Sets the current default settings for the global **RC** structure. In version 2.0 of the Pen API, the **RC** structure is made obsolete by the **HRC** object.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

UINT SetGlobalRC(LPRC *lprc*, **LPSTR** *lpszDefRecog*, **LPSTR** *lpszDefDict* **)**

Parameters

lprc

Address of an **RC** structure or NULL.

lpszDefRecog

Address of string specifying the name of the default recognizer module (maximum 128 bytes).

lpszDefDict

Address of a string specifying the default dictionary path. The list should end with two null characters.

Return Value

Returns the value SGRC_OK if successful. If an error occurs, the return value consists of one or more of the following values, combined using the bitwise-OR operator.

Constant	Description
SGRC_USER	An invalid user name was found in the supplied **RC** structure. The call to **SetGlobalRC** has no effect.
SGRC_PARAMERROR	One or more invalid parameters were detected. The call to **SetGlobalRC** has no effect.
SGRC_RC	The supplied recognition context *lprc* has entries, other than the user name, that contain invalid settings for a global recognition context. The supplied recognition context is ignored.
SGRC_RECOGNIZER	The supplied recognizer module name *lpszDefRecog* is invalid or the recognizer cannot be loaded. The supplied recognizer module name is ignored.
SGRC_DICTIONARY	The supplied dictionary path *lpszDefDict* is invalid or some dictionaries on the path cannot be loaded. The supplied dictionary path is ignored.
SGRC_INIFILE	An error was encountered while saving the new global recognition context

	settings to the pen section of the system registry. The new settings are lost after rebooting Windows.

Comments

Because the default **RC** values are shared among all version 1.0 applications running, the values should be changed only through the Control Panel. Whenever a change is made to the global **RC** values, the WM_PENMISCINFO message is sent to all top-level windows. The *wParam* and *lParam* values are not used, and they are set to 0.

Any of the parameters can be NULL to indicate that the calling application does not want the value changed.

SetGlobalRC uses only the following members of the **RC** structure pointed to by the *lprc* parameter:

clErrorLevel lPcm (PCM_TIMEOUT and PCM_RANGE bits) **lpLanguage lpUser nInkWidth rgbInk wCountry wIntlPreferences wRcDirect wRcPreferences wTimeOut wTryDictionary**

When **InitRC** is called for a new recognizer from within the **SetGlobalRC** call, the **RC** structure that is passed in contains the new values for all members except **hrec** and **rglpdf**. No new recognizer and dictionaries have been set up at this point.

When a version 1.0 application receives a WM_PENMISCINFO message, it should call **ConfigRecognizer** with a WCR_RCCHANGE subfunction request. This should be done for all recognizers that the application has loaded, excluding the default recognizer. The RC Manager calls **ConfigRecognizer** in the new default recognizer with a WCR_RCCHANGE subfunction request.

SetGlobalRC does not save the RCP_MAPCHAR flag in the **wRcPreferences** member of the **RC** structure to the system registry. The RCP_MAPCHAR flag is reflected in the global **RC** for the current session only.

See Also

InitRC, **GetGlobalRC**, **RC**

SetGuideHRC Overview

Overview

2.0

Sets a guide structure into an **HRC** object.

int SetGuideHRC(HRC *hrc*, **LPGUIDE** *lpguide*, **UINT** *nFirstVisible* **)**

Parameters

hrc

Handle to the **HRC** object.

lpguide

Pointer to a **GUIDE** structure, or NULL. All coordinates are in screen coordinates.

nFirstVisible

For boxed controls, *nFirstVisible* refers to the first visible box (leftmost and topmost for left-right, top-down languages like English). For other controls, this is the first visible character position (leftmost for English) in a single-line control, and the first visible line (topmost for English) in multiline controls.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_INVALIDGUIDE	The guide structure is invalid.
HRCR_MEMERR	Insufficient memory.

Comments

This function is useful for doing boxed recognition. The **GUIDE** structure defines the size and position of the boxes. The *nFirstVisible* parameter notifies the recognizer which is the first visible character position (single-line controls) or line (multiline controls) in case the contents were scrolled. The writing direction affects the meaning of this value.

If *lpguide* is NULL, or if all the members in the **GUIDE** structure are 0, the recognizer does not use guides (free input).

See Also

GetGuideHRC, **GUIDE**

SetInternationalHRC

2.0

Sets the country, language, and script direction into a recognition context **HRC**.

int SetInternationalHRC(HRC *hrc*, **UINT** *uCountry*, **LPCSTR** *lpszLangCode*, **UINT** *fuFlags*, **UINT** *uDir* **)**

Parameters

hrc

Handle to the **HRC** object.

uCountry

The country code. A value of 0 indicates that this value should not be set.

lpszLangCode

A three-letter, null-terminated string identifying the language (for example, "enu" or "fra"), or NULL. A value of NULL indicates that the language code should not be changed. For a list of three-letter language identifiers, refer to Volume 1 of the *Programmers Reference* in the Windows Software Development Kit.

fuFlags

Flags. can be either SIH_ALLANSICHAR to indicate the user intends to use the entire ANSI character set, or 0.

uDir

The script direction. This parameter specifies which primary and secondary writing directions to set. The default directions are left to right for the primary direction and top to bottom for the secondary. A value of 0 indicates that the writing direction should not be changed. Possible values for *uDir* are:

Constant	Description
SSH_RD	Left to right and down (English).
SSH_RU	Left to right and up.
SSH_LD	Right to left and down (Hebrew).
SSH_LU	Right to left and up.
SSH_DL	Down and to the left (Chinese).
SSH_DR	Down and to the right (Chinese).
SSH_UL	Up and to the left.
SSH_UR	Up and to the right.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

Setting *fuFlags* to the value of SIH_ALLANSICHAR indicates the recognizer should interpret text written in any language based on ANSI characters. To constrain recognition to a particular language, an application should set *fuFlags* to 0 and provide the appropriate language code in *lpszLangCode*.

SetInternationalHRC overrides the default ALLANSICHAR setting in the recognizer set by **ConfigHREC** for the life of the **HRC** object. **ConfigHREC** should be used to change the default value.

See Also

GetInternationalHRC, **ConfigHREC**

SetMaxResultsHRC Overview

Overview

2.0

Sets the maximum number of guesses a recognizer should make when interpreting pen data. When the recognizer formulates this number of results, the recognition process ends.

int SetMaxResultsHRC(HRC *hrc*, **UINT** *cMaxResults* **)**

Parameters

hrc

Handle to the **HRC** object for the recognizer.

cMaxResults

The maximum number of results a recognition context should generate. This value must be greater than 0.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

Subsequent calls to **SetMaxResultsHRC** override any previous settings. If the application does not call **SetMaxResultsHRC** to explicitly set a maximum value, the default number of results generated is 1.

See Also

GetMaxResultsHRC, **CreateCompatibleHRC**

SetPenAppFlags Overview

Overview

2.0

An application calls this function to set pen-specific properties that apply globally to the application. This function replaces and enhances the **RegisterPenApp** function from version 1.0 of Pen Windows.

void SetPenAppFlags(UINT *fuFlags*, **UINT** *uVersion* **)**

Parameters

fuFlags

Flags specifying application options. The following flags can be combined by using the bitwise-OR operator:

Constant	Description
RPA_HEDIT	Specifies that the system should treat any EDIT-class controls in the application as having HEDIT class.
RPA_KANJIFIXEDBEDIT	Boxed edit controls are a fixed size appropriate for use with kanji. (Japanese version only.)
RPA_DBCSPRIORITY	By default, double-byte equivalents of single-byte characters (as used in Japan) are preferred in recognition results.
RPA_SBCSPRIORITY	By default, single-byte characters are preferred in recognition results. (Japanese version only.)
RPA_DEFAULT	Specifies default pen behavior for the application. This includes RPA_HEDIT.

uVersion

The Pen API version number. The nonzero value PENVER causes the application to be registered with Windows. A value of 0 unregisters the application.

Return Value

This function does not return a value.

Comments

SetPenAppFlags should be called when an application starts with PENVER for the *uVersion* parameter. PENVER is the Pen API version number, defined in PENWIN.H.

PENVER ensures that the structures used are appropriate for the version of the Pen API for which the application was compiled. In version 1.0, *uVersion* was a BOOL value, so by default the version number was 0x0001. Beginning with Pen API version 2.0, PENVER contains the major release number in the HIBYTE and the minor release number in the LOBYTE. Thus, for version 2.0, PENVER is defined as 0x0200.

When an application terminates, it should call **SetPenAppFlags** with *uVersion* set to 0. An application can unregister itself in this way more than once without error to accommodate alternative exit code paths.

An application can call **GetPenAppFlags** to determine which flags were set by an earlier call to **SetPenAppFlags**. However, normally only the system requires this information.

Applications written specifically for Windows 95 and later Windows versions automatically get RPA_DEFAULT so that any edit controls created by such applications become pen-aware.

See Also

GetPenAppFlags

SetPenHook Overview

Overview

1.0 2.0

Installs and removes a pen packet hook. This function is typically used by system-level applications such as Control Panel applications.

BOOL SetPenHook(HKP *hkpOp*, **LPFNRAWHOOK** *lpfn* **)**

Parameters

hkpOp

Operation to be performed. This parameter can be HKP_SETHOOK to install a hook or HKP_UNHOOK to remove a function from the hook list.

lpfn

Pointer to callback function to handle pen packets.

Return Value

Returns TRUE if successful or FALSE if **GetPenInput** is unable to set or remove the hook. The callback function returns FALSE to cancel the processing of a pen packet.

Comments

The pen device generates approximately 100 hardware interrupts per second. At each interrupt, the device sends data to the pen driver, which organizes the data into a *pen packet*. Each packet contains the x- and y-coordinates of the current pen position, the time, and possibly extra OEM data such as pen pressure, angle, and so forth. The pen device may require more than one hardware interrupt to send all the information for a single packet, so the rate at which the driver sends pen packets may be less than the rate of interrupts generated by the pen hardware.

When it has created a pen packet, the driver passes it to the system, which buffers the packets in an internal queue as they arrive from the pen driver. The internal queue is informally known as the "ten-second buffer" to indicate how much data it can hold before overflowing. An application must call **GetPenInput** regularly to remove data from the queue.

SetPenHook enables an application to examine, modify, or cancel pen packets as they arrive from the pen driver before **GetPenInput** sees them.

See Also

SetPenHookCallback, **SetResultsHookHREC**, **GetPenInput**

SetPenHookCallback

1.0 2.0

SetPenHookCallback represents the name of the callback function that the *lpfn* argument of **SetPenHook** points to. An application can use any name.

BOOL lpfn SetPenHookCallback(LPPENPACKET *lppp* **)**

Parameters

lppp

Far pointer to the most recent pen packet received from the pen driver.

Return Value

Returns TRUE to continue processing, FALSE to cancel pen packet.

Comments

For a definition of pen packet, see the description for **SetPenHook**.

At each interrupt, the system adds the latest packet from the pen driver to an internal queue. It then calls the application's **SetPenHookCallback** callback function, providing it with a pointer to the latest packet in the queue. This enables the callback function to examine, modify, or cancel each pen packet as it arrives from the pen driver.

To get the pen packet data from a version 2.0 pen driver, defined as **OEM_PENPACKET**, simply cast the **LPPENPACKET** value passed into this function to the type **LPOEM_PENPACKET**. The pen services detect the pen driver version automatically and return the correct data pen packet data type.

See Also

SetPenHook, **PENPACKET**

SetPenMiscInfo Overview

Overview

1.0 2.0

Sets constants pertaining to the pen system.

LONG SetPenMiscInfo(UINT *wParam*, **LPARAM** *lParam* **)**

Parameters

wParam

Specifies the identifier of the pen system measurement to set. The identifier must be a PMI_ value, and may be combined with PMI_SAVE (to force an immediate initialization file update) using the bitwise-OR operator for some values. See the following table for the possible PMI_ values in *wParam*.

lParam

Specifies the value of the pen system measurement to set. Depending on the value of *wParam* (listed in the first column of the table below), *lParam* can be the address of a structure or a value, as described here:

wParam constant	lParam description
PMI_BEDIT	*lParam* is the address of a **BOXEDITINFO** structure.
PMI_ENABLEFLAGS	*lParam* is a WORD value.
PMI_PENTIP	*lParam* is the address of a **PENTIP** structure.
PMI_TIMEOUT	*lParam* is a UINT value.
PMI_TIMEOUTGEST	*lParam* is a UINT value.
PMI_TIMEOUTSEL	*lParam* is a UINT value.

Return Value

Returns PMIR_OK if successful; otherwise, returns one of the following negative error values:

Constant	Description
PMIR_INDEX	*wParam* is out of range.
PMIR_INIERROR	Error writing to PENWIN.INI file.
PMIR_INVALIDBOXEDITINFO	**BOXEDITINFO** structure is invalid.
PMIR_NA	Support for this value of *wParam* is not available.
PMIR_VALUE	*lParam* is invalid.

Comments

The type of information **SetPenMiscInfo** sets depends on *wParam*. The function is provided for system applications such as Control Panel. User applications should not generally call **SetPenMiscInfo**.

A WM_PENMISCINFO message is posted to all top-level windows whenever **SetPenMiscInfo** successfully changes a setting, forwarding the value for *wParam*. In the case of PMI_BEDIT, a WM_PENMISC message is also broadcast to ensure compatibility with version 1.0 of the Pen API. The *wParam* is set to PMSC_BEDITCHANGE and *lParam* is a far pointer to a **BOXEDITINFO** structure.

SetPenMiscInfo cannot set all the values available in **GetPenMiscInfo** because certain values are determined by the system. These values are PMI_SYSREC, PMI_CXTABLET, PMI_CYTABLET, PMI_SYSFLAGS, PMI_TICKREF, PMI_INDEXFROMRGB, and PMI_RGBFROMINDEX.

The flag PMI_SAVE can be combined with the *wParam* identifier for the following values: PMI_BEDIT, PMI_ENABLEFLAGS, PMI_PENTIP, PMI_TIMEOUT, PMI_TIMEOUTGEST, and PMI_TIMEOUTSEL. This forces Windows to immediately update its initialization information.

Example

The following code sample changes the pen color to red and the time out to a half second (500 milliseconds), then forces a save-file update:

```
PENTIP tip;

GetPenMiscInfo( PMI_PENTIP, (LPARAM)(LPPENTIP) &tip );
tip.rgb = RGB(255, 0, 0);
SetPenMiscInfo( PMI_PENTIP, (LPARAM)(LPPENTIP) &tip );
SetPenMiscInfo( PMI_TIMEOUT | PMI_SAVE, (LPARAM)500 );
```

See Also

GetPenMiscInfo, WM_PENMISCINFO, PMI_

SetRecogHook Overview

Overview

1.0 2.0

Installs and removes a recognition hook. This function works only for Pen API version 1.0 recognizers accessed through **Recognize** or **RecognizeData**.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions. Use **SetResultsHookHREC** instead.

BOOL SetRecogHook(UINT *uScope*, **UINT** *uSetOp*, **HWND** *hwndHook* **)**

Parameters

uScope

Scope of hook. The hook parameter *uSetOp* determines the scope of the hook. The following table lists the HWR_ values for *uSetOp*:

Constant	Description
HWR_RESULTS	The hook window receives a WM_HOOKRCRESULT message before a WM_RCRESULT message is sent to the target window.
HWR_APPWIDE	The hook window receives the message WM_HOOKRCRESULT before a WM_RCRESULT message is sent to the target window if the target window belongs to the same task as the window that set an HWR_APPWIDE hook. This is useful for implementing application-wide gestures. The RCRT_ALREADYPROCESSED flag is set in the **wResultsType** member of the **RCRESULTS** structure sent with WM_RCRESULT if an application-wide hook has already processed the data.

uSetOp

Parameter to determine whether hook is set or removed. The operation parameter *uSetOp* determines whether the hook is set or removed. The following table lists the HKP_ values for *uScope*:

Constant	Description
HKP_SETHOOK	Installs a hook.
HKP_UNHOOK	Removes function from hook list.

hwndHook

Handle to a window.

Return Value

Returns TRUE if successful; otherwise, FALSE.

Comments

SetRecogHook enables a version 1.0 application to examine the results of recognition before they are sent to the target application.

The hook message is WM_HOOKRCRESULT. The *wParam* and *lParam* parameters are the same as for the WM_RCRESULT message. If the window procedure that receives the WM_HOOKRCRESULT message returns FALSE, the message WM_HOOKRCRESULT is not sent to any of the remaining hooks in the chain.

No drawing should occur during the processing of the WM_HOOKRCRESULT and before recognition is complete. Drawing at these times could cause timing problems, with ink reappearing in formerly invisible controls as they are redrawn.

See Also

SetResultsHookHREC

SetResultsHookHREC Overview

Overview

2.0

Sets up a hook callback function for recognition results.

HRECHOOK SetResultsHookHREC(HREC *hrec*, **HRCRESULTHOOKPROC** *lpfnHook* **)**

Parameters

hrec

Module handle of the recognizer library whose results are to be hooked. If *hrec* is set to NULL, the hook function specified in *lpfnHook* receives results from the system default recognizer. If *hrec* is set to SRH_HOOKALL, the hook function receives results for all recognizers the application has installed, including the system recognizer.

lpfnHook

Address of the hook function.

Return Value

Returns a handle to the installed hook if successful; otherwise, the return value is NULL. The application must provide this handle when calling **UnhookResultsHookHREC** to remove the hook.

Comments

An application can set multiple hooks. The system calls the hooks in reverse order –that is, the most-recently-set hook is called first, then the previous hook, and so on. If a hook function captures a result, the function that requested the results returns HRCR_HOOKED to the application.

See Also

ResultsHookHREC, **UnhookResultsHookHREC**

SetStrokeAttributes Overview

Overview

2.0

Sets attributes of a stroke or of a class of strokes in an **HPENDATA** object.

int SetStrokeAttributes(HPENDATA *hpndt*, **UINT** *iStrk*, **LPARAM** *lParam*, **UINT** *uOption* **)**

Parameters

hpndt

Handle to the **HPENDATA** object.

iStrk

Zero-based stroke index. A value of IX_END can be used to specify the last available stroke in the pen data.

lParam

A pointer to a structure (cast to the LPARAM type), or a byte, word, or double-word value, depending on *uOption*. This parameter cannot be NULL.

uOption

Specifies the attributes to set. This parameter has one of the following values:

Constant	Description
SSA_DOWN	Set the up and down state of the pen tip for the stroke specified by *iStrk*. *lParam* is nonzero to make it a downstroke or 0 to make it an upstroke.
SSA_PENTIP	Set the pen-tip characteristics (color, width, nib type) used by the stroke specified by *iStrk*. *lParam* is a pointer to a **PENTIP** structure. If this attribute does not already exist in the stroke class table, a new entry for this type of stroke is created. There can be up to 255 different types of strokes.
SSA_PENTIPCLASS	Set the pen-tip characteristics (color, width, nib) for all strokes of which the stroke specified by *iStrk* is a member. *lParam* is a pointer to a **PENTIP** structure. If the new type already exists in the stroke class table, the types are merged.
SSA_SELECT	Set the selection status of the stroke specified by *iStrk*. *lParam* is nonzero to select it or 0 to deselect it.
SSA_TIME	Set the absolute time of the stroke. *lParam* is a pointer to an **ABSTIME** structure. The **sec** member of the **ABSTIME** structure specifies the number of seconds since Jan 1, 1970, and the **ms** member specifies the number of milliseconds offset from that time to the beginning of the stroke.
SSA_USER	Set the user value for the stroke specified

	by *iStrk*. *lParam* is a BYTE, WORD, or DWORD value, and the pen data must have been created with the corresponding size allocated for user values.
SSA_USERCLASS	Set the user value for the class of strokes of which the stroke specified by *iStrk* is a member. *lParam* is a BYTE, WORD, or DWORD value, and the pen data must have been created with the corresponding size allocated for user values.

Return Value

Returns PDR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Memory error.
PDR_PNDTERR	Invalid pen data.
PDR_SCTERR	Stroke class table may be full, or related error.
PDR_STRKINDEXERR	Invalid stroke index.
PDR_TIMESTAMPERR	Timing information was removed.
PDR_VERSIONERR	Could not convert old pen data.

Comments

The bounding rectangle of the pen data is recalculated each time the SSA_DOWN option is used, because the rectangle represents the bounds of only the pen-down points. Setting a pen-up point to the down state simply adds (union) the bounding rectangles of the existing pen data and the stroke. Setting a pen-down point to the up state is more calculation-intensive, however, since the bounding rectangle must be calculated from all of the remaining strokes.

See Also

CreatePenDataEx, **GetStrokeAttributes**, **GetStrokeTableAttributes**, **SetStrokeTableAttributes**, **PENTIP**

SetStrokeTableAttributes Overview

Overview

2.0

Sets attributes of a stroke's class within an **HPENDATA** object. (The class is an entry in a table stored in the **PENDATAHEADER** structure. Modifying the table entry affects all the strokes described by the entry.)

int SetStrokeTableAttributes(HPENDATA *hpndt*, **UINT** *iTblEntry*, **LPARAM** *lParam*, **UINT** *uOption* **)**

Parameters

hpndt

Handle to the **HPENDATA** object.

iTblEntry

Zero-based table index to the class entry in the pen data header.

lParam

A pointer to a structure (cast to the LPARAM type), or a byte, word, or doubleword value, depending on *uOption*. This parameter cannot be NULL.

uOption

Specifies the attributes to set. This parameter can be one of the following:

SSA_PENTIPTABLE

Set the pen-tip characteristics (color, width, nib) of the class of strokes specified by *iTblEntry*. *lParam* is a pointer to a **PENTIP** structure. All the strokes sharing this entry in the stroke class table receive the new pen-tip attribute.

SSA_USERTABLE

Set the user value, if any, of the class of strokes specified by *iTblEntry*. *lParam* is a byte, word, or doubleword value, and the pen data must have been created with the corresponding size allocated for user values. All the strokes sharing this stroke class table entry receive the new user value.

Return Value

Returns PDR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
PDR_COMPRESSED	Pen data is compressed.
PDR_ERROR	Parameter or other unspecified error.
PDR_MEMERR	Memory error.
PDR_PNDTERR	Invalid pen data.
PDR_SCTERR	Stroke class table may be full, or related error.
PDR_VERSIONERR	Could not convert old pen data.

See Also

CreatePenDataEx, **GetStrokeAttributes**, **GetStrokeTableAttributes**, **SetStrokeAttributes**, **PENTIP**

SetWordlistCoercionHRC Overview

Overview

2.0

Specifies to what degree input must match a word list set into an **HRC**. **SetWordlistCoercionHRC** determines the influence a recognizer's word list or dictionary has on the recognizer's guesses.

int SetWordlistCoercionHRC(HRC *hrc*, **UINT** *uCoercion* **)**

Parameters

hrc

Handle to the **HRC** object.

uCoercion

Coercion flag. This can be one of the following:

SCH_ADVISE

The word list serves only to advise the recognizer, but lacks a strong degree of influence. Recognition results are not strongly coerced to match the word list.

SCH_FORCE

If the recognizer's guess is not found in the word list, the closest matching entry in the list is returned. For example, if the recognizer interprets writing as "Cana", it returns "Canada" from a word list of country names.

SCH_NONE

Do not coerce. This flag can be used to turn off a previous request.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error, including an attempt to set coercion with no word lists set into the recognition context.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

The default type of coercion a recognizer provides is SCH_ADVISE. That is, results are not strongly coerced to any word list that might be set into a recognition context.

Coercion is used only if a word list (**HWL**) has actually been set into an **HRC** with **SetWordlistHRC**, or if the recognizer's dictionary is enabled by **EnableSystemDictionaryHRC**. If the **HRC** is configured with a word list and the recognizer's dictionary is also enabled, coercion is done on both; the priority depends on the recognizer.

See Also

CreateHWL, **GetWordlistCoercionHRC**

SetWordlistHRC Overview

Overview

2.0

Sets a word list into a recognition context **HRC** object.

int SetWordlistHRC(HRC *hrc*, **HWL** *hwl* **)**

Parameters

hrc

Handle to the **HRC** object.

hwl

Handle to a word list to use, or NULL. A value of NULL means that the recognizer should not constrain recognition based on any word list, including its own dictionary.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

Specifying NULL for *hrc* does not destroy the word list specified by *hwl*. Applications must call **DestroyHWL** to destroy a word list.

Only one word list can be set into an **HRC** at a time. This is independent of the recognizer's dictionary, which can be manipulated through the function **EnableSystemDictionaryHRC**.

For a description of word lists and how a recognizer uses them, see "Configuring the HRC" in Chapter 5, "The Recognition Process."

See Also

CreateHWL, **DestroyHWL**

ShowKeyboard Overview

Overview

1.0 2.0

Shows or hides the on-screen keyboard. (Not supported in Japanese version.)

Note This function is provided only for compatibility with version 1.0 of the Pen API, and will not be supported in future versions. It is not supported for 32-bit applications. Applications should interface directly with the on-screen keyboard.

BOOL ShowKeyboard(HWND *hwnd*, **UINT** *wCommand*, **LPPOINT** *lppt*, **LPSKBINFO** *lpSKBInfo* **)**

Parameters

hwnd

Handle of window invoking the on-screen keyboard.

wCommand

A show request and optional keypad. The values for the show requests are listed in the "Comments" section below.

lppt

Address of a **POINT** structure containing the keyboard position in screen coordinates. If NULL, the keyboard appears centered on the display.

lpSKBInfo

Address of an **SKBINFO** structure to be filled with values for the current keyboard. This parameter is ignored if NULL. If the **hwnd** member of the **SKBINFO** structure is NULL, no on-screen keyboard has been loaded.

Return Value

Returns TRUE if successful; otherwise FALSE.

Comments

Any user action on the keyboard itself overrides the function requests. For example, if the user closes the on-screen keyboard, the keyboard becomes unregistered for all windows in all applications. If the user minimizes the keyboard, the active **SKBINFO** structure is changed to reflect the new state.

ShowKeyboard tracks registration information for up to 20 window handles. If one application displays the keyboard and then another one does the same thing, both applications must request that the keyboard be hidden before it actually disappears.

The following SKB_ requests can be specified in the *wCommand* parameter:

Constant	Description
SKB_HIDE	Hides the on-screen keyboard. This request may not actually hide the keyboard if another application is also using it. The command decrements the use count for the keyboard. SKB_HIDE automatically loads the on-screen keyboard if it is not already present.

SKB_QUERY	Returns the current state of the keyboard pointed to by the *lpSKBInfo* parameter without invoking a new keyboard state. This command does not automatically load the on-screen keyboard.
SKB_SHOW	Shows the on-screen keyboard in a restored state at the most recently used screen location. This command increments a window-use count. SKB_SHOW automatically loads the on-screen keyboard if it is not present.

The SKB_SHOW command in the *wCommand* parameter can be combined using the bitwise-OR operator with any of the command or keypad requests listed in the following tables:

Constant	**Description**
SKB_CENTER	Centers the keyboard on the display. This command has higher priority than SKB_MOVE.
SKB_MINIMIZE	Displays the on-screen keyboard in a minimized state. This command can be used with SKB_CENTER or SKB_MOVE. If it is used with SKB_MOVE, the location specified will be used when the keyboard is restored.
SKB_MOVE	Moves the keyboard to the location specified by the *lppt* parameter. If *lppt* is NULL, the keyboard is centered on the screen. If it is not NULL, *lppt* specifies a pointer to the x and y screen coordinates of the upper-left corner of the restored keyboard.

The following keypad requests can be used with SKB_SHOW in the *wCommand* parameter. The SKB_BASIC, SKB_FULL, and SKB_NUMPAD constants can not be combined with the OR operator:

Constant	**Description**
SKB_BASIC	Switches the keyboard to a partial keyboard with no extended keys.
SKB_FULL	Switches the keyboard to the full 101-key display.
SKB_NUMPAD	Switches the keyboard to a partial keyboard consisting only of ESC, TAB, SHIFT, and the numeric keypad.

The following three bitmaps are provided for owner-draw push buttons that can be used to invoke the on-screen keyboard. The application must process WM_DRAWITEM and other button-related code. On-screen keyboard push buttons should behave the same way as other standard buttons (for example, the Minimize button) and take the appropriate action when a button-up message is received following a button-down message.

Constant	**Description**
OBM_SKBBTNUP	Push button is up.

OBM_SKBBTNDOWN Push button is down.
OBM_SKBBTNDISABL Push button is disabled.
ED

Example

The up bitmap, for example, can be loaded as shown in the following code sample:

```
HANDLE hDLL = GetSystemMetrics( SM_PENWINDOWS );
HBITMAP hBitmap = LoadBitmap( hDLL,
                        MAKEINTRESOURCE(OBM_SKBBTNUP) );
```

The application must call the Windows **DeleteObject** function to delete each bit-map handle returned by the Windows **LoadBitmap** function.

The button should be left in the up state after it is released. If the user closes the keyboard and the buttons are up, they will still be up the next time the keyboard is opened. The following code sample retrieves the current keyboard and restores the current state:

```
#include <penwin.h>

if (ShowKeyboard( hwnd, SKB_SHOW, NULL, NULL)) // Nonzero: no error.
{
            .
            .                          // Perform some tasks.
            .
     ShowKeyboard( hwnd, SKB_HIDE, NULL, NULL );
}
else
     ErrorMsg( "Unable to use Screen Keyboard" );
```

The following code sample moves the keyboard and then puts it back into its starting position:

```
SKBINFO skbinfo;
WORD wCommand = SKB_SHOW | SKB_MOVE;
POINT pnt;

pnt.x = wSKBLeft;    // Initialize point.
pnt.y = wSKBTop;

// Show the keyboard.

ShowKeyboard( hwnd, wCommand, &pnt, &skbinfo );
            .
            .
            .
// Now restore the keyboard.

if (skbinfo.fVisible)
     wCommand = SKB_SHOW | SKB_MOVE |
           (skbinfo.fMinimized ? SKB_MINIMIZED : 0);
else
     wCommand = SKB_HIDE;

ShowKeyboard( hwnd, wCommand, (LPPOINT)(&skbinfo.rect), NULL) ;
```

StartInking Overview

Overview

2.0

Starts inking feedback while pen input is being collected.

int StartInking(HPCM *hpcm*, **UINT** *idEvent*, **LPINKINGINFO** *lpinkinginfo* **)**

Parameters

hpcm

Handle to the current collection. This is the return value from **StartPenInput**.

idEvent

The identifier of the packet at which to start inking.

lpinkinginfo

Address of an **INKINGINFO** structure, used to specify the characteristics of the ink. This parameter can be NULL to use the default ink characteristics. Otherwise, the structure's **cbSize** member must be initialized with sizeof(INKINGINFO).

Return Value

Returns PCMR_OK if inking started successfully; otherwise, returns one of the following:

Constant	Description
PCMR_DISPLAYERR	There is no display device, or it was unable to ink at this time, or there was an error in setting the pen-tip characteristics.
PCMR_ERROR	The **INKINGINFO** structure contains invalid values, or there was some other unspecified error.
PCMR_INVALIDCOLLECTION	The *hpcm* handle is invalid because the calling application did not start input by calling **StartPenInput**.
PCMR_INVALID_PACKETID	The *idEvent* parameter is invalid.

Comments

An application calls **StartInking** to track pen movement while the pen tip is down. When pen input is started by calling the **StartPenInput** function, Windows initializes the internal **INKINGINFO** structure as follows:

- The **wFlags** member is set to PII_INKPENTIP | PII_INKCLIPRECT.
- The **tip** member is set to the system default pen tip, as obtained by calling the **GetPenMiscInfo** function.
- The **rectClip** member is set to the client area, in screen coordinates, of the window that was used in the call to the **StartPenInput** function.

The first call to **StartInking** with the *lpinkinginfo* parameter set to NULL starts inking with the settings listed above. If the calling application uses a non-NULL value for *lpinkinginfo*, the appropriate internal

inking parameters are modified before inking starts, depending on the flags set in the **wFlags** member of the **INKINGINFO** structure.

Whenever **StartInking** is called, the current settings of the internal inking structure are added to or replaced. Specific values must be set in the members of **INKINGINFO** to disable them. Refer to the description of each member in the **INKINGINFO** structure for these values.

If a region is passed in for clipping or stopping the ink, the application must destroy the region. Since a copy is made, the region can be destroyed immediately following the call to **StartInking**. The application can specify either a clip region or a clip rectangle. Specifying both will result in the clip rectangle being ignored.

Example

The following code example changes the inking tip from the default (as set by a call to **StartPenInput**) to red ink, 5 pixels wide. It also adds an *inkstop* rectangle (inking stops if the pen touches down inside the inkstop rectangle). The clipping rectangle remains unchanged from the default settings.

```
INKINGINFO inkinginfo;

inkinginfo.cbSize           = sizeof( INKINGINFO );
inkinginfo.wFlags           = PII_INKPENTIP | PII_INKSTOPRECT;
inkinginfo.tip.cbSize       = sizeof( PENTIP );
inkinginfo.tip.rgb          = RGB( 255,0,0 );
inkinginfo.tip.bwidth       = 5;
inkinginfo.rectInkStop.left = rectInkTop.top = 0;
inkinginfo.rectInkStop.right = rectInkTop.bottom = 100;

ClientToScreen( hwnd, (LPPOINT)&(inkinginfo.rectInkStop) );
ClientToScreen( hwnd, (LPPOINT)&(inkinginfo.rectInkStop.right) );
StartInking( hpcm, wEventRef, &inkinginfo );
```

See Also

INKINGINFO, **StartPenInput**, **StopInking**

StartPenInput Overview

Overview

2.0

Begins collecting information from the pen input stream.

HPCM StartPenInput(HWND *hwnd*, **UINT** *idEvent*, **LPPCMINFO** *lppcmInfo*, **LPINT** *lpiErrRet* **)**

Parameters

hwnd

Handle of the window that receives the WM_PENEVENT messages generated by **StartPenInput**.

idEvent

Identifies the packet in the global queue of pen packets maintained internally by the system. The *idEvent* is the low-order word of the value returned from the **GetMessageExtraInfo** function when processing a WM_LBUTTONDOWN message. For a definition of pen packet, see the description for **SetPenHook**.

lppcmInfo

Address of a **PCMINFO** structure. If NULL, the system creates a default **PCMINFO** structure with the following values:

Constant	Description
dwPcm	PCM_RECTBOUND | PCM_TIMEOUT | PCM_TAPNHOLD
rectBound	The bounding rectangle of the window identified by *hwnd*
	These values determine that the input session (a) terminates when pen activity ceases for a specified time-out period; (b) terminates when the pen moves outside the bounds of the window; or (c) does not begin at all if the user taps and holds the pen for a specified time-out period (about one-half second). This "tap-and-hold" gesture switches the system from input mode to selection mode. Usually, the cursor changes from a pen (indicating input) to an upside-down arrow (indicating selection) to acknowledge the switch. Subsequent pen movement then behaves as a mouse with the left button held down. This allows the user to make selections as though dragging the mouse.

lpiErrRet

Address of an integer that receives a return code when **StartPenInput** terminates. If NULL, no return code is provided. If not NULL, the return code is one of the following values:

Constant	Description
PCMR_OK	Pen collection was successfully started.

PCMR_ALREADYCOLLECTING	**StartPenInput** has already been called for this session.
PCMR_ERROR	Illegal parameter or unspecified error.
PCMR_INVALID_PACKETID	Invalid *idEvent* parameter.
PCMR_SELECT	Tap-and-hold gesture detected. Collection is not started, as described in the description of the *lppcmInfo* parameter.
PCMR_TAP	The pen has briefly tapped the tablet. This event may be inadvertent and in any case does not indicate that the user has started to write; therefore, collection is not started.

Return Value

Returns a handle to the application's queue of pen packets, if successful. Returns NULL to indicate an error or the detection of a tap or press-and-hold condition.

Comments

When this function returns successfully, Windows creates a queue of pen packets for the calling application. All subsequent pen packets from the pen device, beginning with the packet identified by the *idEvent* argument, are placed into the queue. Until a termination condition occurs (as specified in the *lppcmInfo* parameter), or until the application calls **StopPenInput**, the queue continues to receive all the packets generated by the pen device as the pen moves.

An application can retrieve all the pen input in its queue of pen packets but should never destroy the queue.

In event mode (the default mode), the collection session specified by the *hpcm* of the **GetPenInput** function becomes invalid when the WM_PENEVENT message (with the PE_TERMINATED submessage) is removed from the application's message queue. This message is posted to the application's message queue either as a consequence of automatic termination or a call to **StopPenInput**.

In polling mode, the application's queue of pen packets is destroyed (and the *hpcm* of **GetPenInput** becomes invalid) after a successful call to **StopPenInput** or a termination return value from the **GetPenInput** function.

If *lppcmInfo* is NULL, a default **PCMINFO** structure is established with the **dwPcm** member set to PCM_RECTBOUND | PCM_TIMEOUT | PCM_TAPHOLD, the **rectBound** member set to the bounds of *hwnd*, and the **dwTimeout** member set to the default system time out.

If the **dwPcm** member of **PCMINFO** does not have the PCM_DOPOLLING flag set, WM_PENEVENT messages are sent to the specified window for significant events such as pen down, pen up, or after some threshold number of points has been received. Otherwise, the application should poll for data using **GetPenInput**.

Other bits in the **dwPcm** member of **PCMINFO** can be used to determine which conditions, if any, terminate pen input. An application can also call **StopPenInput** to explicitly terminate the input.

Example

The following example initiates pen input in a window procedure on detection of pen down:

```
static HPCM vhpcm;
//... omitted ...

switch (message)
{

case WM_LBUTTONDOWN:
     {

           // Get extra info associated with event:

           DWORD dwExtraInfo = GetMessageExtraInfo();

           if (IsPenEvent( message, dwExtraInfo ))   // Checks PDK bits
           {
                 PCMINFO pcminfo;        // Pen collection mode structure

                 pcminfo.cbSize    = sizeof( PCMINFO );
                 pcminfo.dwPcm     = PCM_RECTBOUND | PCM_TIMEOUT;
                 pcminfo.dwTimeout = dwTimeOutDefault;   // 1 second

                 // Set inclusion rect to client area, but in screen coords:

                 GetClientRect( hwnd, &pcminfo.rectBound );
                 ClientToScreen( hwnd, (LPPOINT) &pcminfo.rectBound );
                 ClientToScreen( hwnd, (LPPOINT) &pcminfo.rectBound.right );

                 // Start gathering input:

                 if (vhpcm = StartPenInput( hwnd,
                            LOWORD( dwExtraInfo ), &pcminfo, NULL ))
                       return 1L;                          // We handled it
           }

           // Else fall into DefWindowProc below...

     }
     break;
```

See Also

GetPenInput, **StopPenInput**, **PCMINFO** WM_PENEVENT, PCM_

StopInking Overview

Overview

2.0

Stops inking feedback.

int StopInking(HPCM *hpcm* **)**

Parameters

hpcm

Handle to the current collection. This is the return value from **StartPenInput**.

Return Value

Returns PCMR_OK if successful; otherwise, returns the following value:

Constant	Description
PCMR_INVALIDCOLLECTION	The *hpcm* handle is invalid, or there is no collection, or inking has not been started.

Comments

Inking must have been started by using the **StartInking** function for this function to have any effect.

See Also

StartInking

StopPenInput Overview

Overview

2.0

Terminates collection of pen input.

int StopPenInput(HPCM *hpcm*, **UINT** *idEvent*, **int** *nTermReason* **)**

Parameters

hpcm

Handle to the collection of the pen data gathered during the input session. **HPCM** stands for "handle to a pen collection mode."

idEvent

The identifier of the packet in the task-specific queue at which the pen input should be terminated. If this value is PID_CURRENT, pen input stops immediately (that is, at the latest position in the task queue) and no further input is collected. The *idEvent* parameter is the low-order word of the value returned from the Windows **GetMessageExtraInfo** function when processing a WM_LBUTTONDOWN message.

nTermReason

The reason for termination. This value is passed to the termination message PE_TERMINATED. It can be one of the following:

Constant	Description
PCMR_APPTERMINATED	Application terminated input.
PCMR_TERMBOUND	Pen was pressed outside bounding rectangle or region.
PCMR_TERMEX	Pen was pressed inside exclusion rectangle or region.
PCMR_TERMPENUP	Pen was lifted from the tablet.
PCMR_TERMRANGE	Pen left the tablet's range of sensitivity.
PCMR_TERMTIMEOUT	Time-out expired.

Return Value

Returns PCMR_OK if successful; otherwise, the return value can be one of the following:

Constant	Description
PCMR_INVALIDCOLLECTION	The *hpcm* handle is invalid because the calling application did not start input with **StartPenInput**.
PCMR_INVALID_PACKETID	*idEvent* is invalid.

Comments

This function allows an application to explicitly terminate pen collection without waiting for one of the conditions specified by **StartPenInput** in the **dwPcm** member of **PCMINFO**.

Due to the asynchronous nature of pen input messages, the application should wait for the WM_PENEVENT message with *wParam* set to PE_TERMINATED to make sure that the pen input

process has completely terminated. This does not apply if the application is using the polling method of pen input.

See Also

StartPenInput

SymbolToCharacter Overview

Overview

1.0 2.0

Converts an array of SYV_ symbol values to an ANSI string.

BOOL SymbolToCharacter(LPSYV *lpsyv*, **int** *cSyv*, **LPSTR** *lpstr*, **LPINT** *lpnConv* **)**

Parameters

lpsyv

Address of the array of SYV_ symbol values.

cSyv

Count of symbols in the *lpsyv* array, including the terminating SYV_NULL.

lpstr

Address of a buffer that receives the ANSI string. The buffer should be large enough to hold at least *cSyv* number of ANSI characters (including SYV_NULL).

lpnConv

If not NULL, *lpnConv* contains the number of symbols converted when the function returns. If NULL, this parameter is ignored.

Return Value

Returns TRUE if successful. If one or more symbols cannot be converted to ANSI, the return value is FALSE.

Comments

For ANSI characters, the size of the *lpstr* buffer must be at least *cSyv* bytes. For double-byte characters (kanji, for example), the buffer size must be at least (2 * *cSyv*) bytes. The **SymbolToCharacter** function converts at most *cSyv* symbol values from *lpsyv* and places the equivalent ANSI characters in the *lpstr* buffer. The conversion proceeds until an SYV_NULL value is encountered or until *cSyv* symbols have been converted. An SYV_NULL is converted to 0. The actual number of symbols converted is returned in *lpnConv* if *lpnConv* is not NULL.

See Also

CharacterToSymbol, **SYG**, SYV_

TargetPoints Overview

Overview

2.0

Determines the target to which pen data belongs.

int TargetPoints(LPTARGINFO *lptarginfo*, **LPPOINT** *lppt*, **DWORD** *dwReserved*, **UINT** *fuReserved*, **LPSTROKEINFO** *lpsi* **)**

Parameters

lptarginfo

Address of a targeting data **TARGINFO** structure.

lppt

Address of a buffer of **POINT** structures in tablet coordinates.

dwReserved

This parameter is reserved for future use and its value is ignored.

fuReserved

This parameter is reserved for future use and its value is ignored.

lpsi

A pointer to a **STROKEINFO** structure. This structure holds information about the stroke being targeted.

Return Value

Returns an array index, starting from 0, of the target in the **rgTarget** array of the **TARGINFO** structure, if successful. If no suitable target is found, or if there are no points to target, **TargetPoints** returns -1.

Comments

To select the desired targeting behavior, the application should set the **dwFlags** member of the **TARGINFO** structure that *lptarginfo* points to.

See Also

GetPenInput, **TARGET**, **TARGINFO**

TPtoDP Overview

Overview

1.0 2.0

Converts points in tablet coordinates to display (screen) coordinates.

BOOL TPtoDP(LPPOINT *lppt*, **int** *cPnt* **)**

Parameters

lppt

Address of an array of **POINT** structures to convert to display coordinates. This parameter cannot be NULL.

cPnt

Number of **POINT** structures to convert.

Return Value

Returns TRUE if the conversion was successful; otherwise, returns FALSE.

Comments

The conversion fails if some tablet points lie outside the region mapped to the screen.

Because of rounding errors, the **DPtoTP** and **TPtoDP** functions are not guaranteed to be perfect inverses of each other.

See Also

DPtoTP

TrainContext Overview

Overview

1.0 2.0

Provides the recognizer a previous recognition result that may contain errors, plus the correct interpretation of the raw data.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions. Use **TrainHREC** instead.

BOOL TrainContext(LPRCRESULT *lprcresult*, **LPSYE** *lpsye*, **int** *csye*, **LPSYC** *lpsyc*, **int** *csyc* **)**

Parameters

lprcresult

Address of the **RCRESULT** structure containing the handle to the pen data that contains the raw data and the recognizer's original interpretation of that data. This parameter cannot be NULL.

lpsye

Address of an array of **SYE** structures that specify the correct interpretation of the raw data. The values of the **iSyc** members of these structures index the **SYC** structures pointed to be the *lpsyc* parameter.

csye

The number of **SYE** structures in the *lpsye* array.

lpsyc

An array of **SYC** structures that establish the mapping between the raw data and the characters in the **hpendata** member of the structure pointed to by the *lprcresult* parameter.

csyc

The number of **SYC** structures in the *lpsyc* array.

Return Value

Returns TRUE if the ink is accepted for training; otherwise, returns FALSE.

Comments

TrainContext is called by an application with a recognition result that may contain mistakes, along with a correct interpretation, so that the recognizer can learn from the mistake and improve subsequent recognition. A second, simpler training function for 1.0 recognizers is provided by **TrainInk**.

TrainContext internally calls the function **TrainContextInternal** exported by the recognizer identified by the **hrec** member of the **RC** structure pointed to by the **lprc** member of the **RCRESULT** structure. A version 1.0 recognizer should export both **TrainContextInternal** and **TrainInkInternal**, but can simply return FALSE from both functions if the recognizer does not support this type of training.

When a training application is able to provide contextual information (such as segmentation suggestions) to the version 1.0 recognizer, it calls the **TrainContext** function. The *lprcresult* parameter points to an **RCRESULT** structure that contains the results of a previous recognition. The raw data is also contained in the **hpendata** member of the structure pointed to be *lprcresult*.

In addition to providing the incorrect interpretation of the data (by means of the symbol graph, the **lpsyv**

member in the **RCRESULT** structure), a more detailed, correct interpretation is also provided by the **SYE** structures and **SYC** structures. Because the correct interpretation is passed by **SYE** structures, it is possible to suggest segmentation boundaries to the recognizer.

Suppose, for example, that a user writes "lc," and the recognizer interprets it as "k". A trainer calls **TrainContext** using, first, an array of **SYC** structures that point to the ink of the "lc" and, second, the two **SYE** structures with the **SYV** values "l" and "c". These two **SYE** structures share the same index into the *lpsyc* array, indicating that both use the ink that was interpreted as "k".

Segmentation errors can be corrected in the other direction as well. Suppose, for example, the user writes "k" and the recognizer interprets it as "lc". A trainer could call **TrainContext** using a single **SYE** with **SYV** values equal to "k" and an array of **SYC** structures that incorporate the ink the recognizer had previously assigned to the "l" and the "c".

To train several **SYV** symbol values to a single piece of ink (for example, a long stroke that is an "he" ligature), there will be two consecutive **SYE** structures—one for the "h" and one for the "e". Both **SYE** structures have the same **iSyc** member; this means that the **SYE** structures both point to the same ink. A recognizer must take this into consideration to avoid training the two characters separately using the same ink for both; that would result in having "he" trained as "he he".

A recognizer can supply its own custom training dialog boxes. An application should check whether the recognizer supports custom training by calling **ConfigRecognizer** with the WCR_TRAIN subfunction.

The trainer does not display an error message if **TrainInk** or **TrainContext** returns FALSE. Error messages that occur when training fails must be handled by the recognizer.

See Also

ConfigRecognizer, **TrainInk**, **TrainHREC**, **SYC**, **SYE**, SYV_

TrainHREC Overview

Overview

2.0

Passes ink and its symbol interpretation to the recognizer for training.

int TrainHREC(HREC *hrec*, **LPSYV** *lpsyv*, **UINT** *cSyv*, **HPENDATA** *hpndt*, **UINT** *uConflict* **)**

Parameters

hrec

Module handle of the recognizer library. If this value is NULL, the system default recognizer is used.

lpsyv

Address of an array of symbols to train.

cSyv

Count of symbols in *lpsyv*. This must be greater than 0.

hpndt

Handle to an **HPENDATA** object.

uConflict

One of the following TH_ values that specify how to handle training conflicts:

Constant	Description
TH_QUERY	Query the user if the proposed training conflicts with symbols in the database.
TH_FORCE	Perform the training without querying the user, even if there is a conflict with the database.
TH_SUGGEST	Abandon the training if there is any conflict with the database and return an error (HRCR_CONFLICT).

Return Value

Returns HRCR_OK if training is successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_CONFLICT	TH_SUGGEST was specified but there was a conflict with the database. No training was done.
HRCR_INVALIDPNDT	Invalid **HPENDATA** object.
HRCR_MEMERR	Insufficient memory.
HRCR_UNSUPPORTED	The recognizer does not support this function.

Comments

Typically, an application calls **TrainHREC** to train a single symbol. In other words, *lpsyv* points to a single symbol that is followed by an SYV_NULL terminator. However, multiple symbols—for example, those

representing the character string "ng"—may also be trainable, depending on the recognizer.

If *uConflict* is TH_QUERY, the recognizer is free to prompt the user with a dialog box to resolve training conflicts. If it is TH_FORCE, the training is performed regardless of conflicts and the original conflicting data may be lost. TH_SUGGEST trains the recognizer only if there are no conflicts; otherwise, the call fails and returns HRCR_CONFLICT.

If the user picks a meaning for some ink from a list of alternatives, such as in a boxed edit control, the application can elect to train the recognizer with this information. In this case, either TH_FORCE or TH_SUGGEST is a suitable value for *uConflict*.

Training gestures depends on the recognizer. The Microsoft Handwriting Recognizer (GRECO.DLL) does not support training for gestures.

See Also

CreateCompatibleHREC

TrainInk Overview

Overview

1.0 2.0

Provides raw data and a correct interpretation of the data to the recognizer.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions. Use **TrainHREC** instead.

BOOL TrainInk(LPRC *lprc*, **HPENDATA** *hpndt*, **LPSYV** *lpsyv* **)**

Parameters

lprc

Address of an **RC** structure, or NULL. If this parameter is NULL, the RC Manager replaces it with a pointer to the global **RC** structure, then calls the recognizer associated with the global **RC**. If *lprc* is not NULL, the RC Manager calls the recognizer identified by the **hrec** member of the **RC** structure.

hpndt

Handle to an **HPENDATA** object containing the ink to be trained. This parameter cannot be NULL.

lpsyv

Pointer to a string of **SYV** symbol values terminated by SYV_NULL. This parameter cannot be NULL.

Return Value

Returns TRUE if the ink described by the pen data could be trained; otherwise, it returns FALSE.

Comments

Applications call **TrainInk** with raw data accompanied by a correct interpretation of the data, so that the recognizer can improve subsequent recognition. A second, more complex training function for version 1.0 recognizers is provided by **TrainContext**.

TrainInk is called by an application to access the **TrainInkInternal** function in the recognizer library. A private 1.0 recognizer must export both **TrainInkInternal** and **TrainContextInternal**, but the functions can simply return FALSE if the recognizer does not support this type of training.

TrainInk provides the lowest level of basic shape training. It requests the recognizer to assign the meaning in *lpsyv* to the ink in *hpndt*. The recognizer should interpret the ink to meet that request.

In the most common case, *lpsyv* points to a single character, and the recognizer will train a new shape based on the ink and that character. In other cases, multiple **SYV** symbol values can be passed, indicating that the ink represents multiple characters. The recognizer must decide whether to simply add a new shape with a meaning based on multiple **SYV** symbol values or to segment the ink into separate shapes for each **SYV**.

An application should check whether a recognizer supports training by calling **ConfigRecognizer** with the WCR_TRAIN subfunction.

The trainer does not display an error message if **TrainInk** or **TrainContext** returns FALSE. Error messages that occur when training fails must be handled by the recognizer.

See Also

ConfigRecognizer, **TrainContext**, **TrainHREC**, SYV_

TrimPenData Overview

Overview

2.0

Removes selected data from an **HPENDATA** object.

HPENDATA TrimPenData(HPENDATA *hpndt*, **DWORD** *dwTrimOptions*, **DWORD** *dwReserved* **)**

Parameters

hpndt

Handle to the **HPENDATA** object.

dwTrimOptions

The following option flags are listed in the order in which the trimming operations are performed. For example, OEM data is removed (TPD_OEMDATA) before duplicate points (TPD_COLLINEAR).

Constant	Description
TPD_RECALCSIZE	Recalculate size of pen data and reallocate if smaller.
TPD_UPPOINTS	Remove pen-up strokes from the **HPENDATA** object.
PHW_PRESSURE	Remove OEM pressure information.
PHW_HEIGHT	Remove OEM height information.
PHW_ANGLEXY	Remove OEM XY-angle information.
PHW_ANGLEZ	Remove OEM Z-angle information.
PHW_BARRELROTATION	Remove OEM barrel rotation information.
PHW_OEMSPECIFIC	Remove OEM-specific value information.
PHW_PDK	Remove per-point Pen Driver Kit (PDK_) information.
TPD_PHW	Remove all OEM and PDK information, but not stroke tick or user data.
TPD_OEMDATA	Remove all OEM values and PDK data.
TPD_PENINFO	Remove **PENINFO** structure from header. Note that any OEM information present is discarded.
TPD_COLLINEAR	Remove collinear and duplicate (coincident) points. There may not be any OEM data.
TPD_USER	Remove per-stroke user information.
TPD_TIME	Remove per-stroke timing information.
TPD_EMPTYSTROKES	Remove all strokes with 0 points.
TPD_EVERYTHING	Remove everything possible except pen-down strokes. This includes both TPD_ and PHW_ flags.

dwReserved

Must be 0.

Return Value

Returns PDR_OK if successful; otherwise, it returns one of the following negative values:

Constant	Description
PDR_COMPRESSED	The pen data was compressed.
PDR_ERROR	An unspecified memory error occurred.
PDR_MEMERR	Memory error.
PDR_OEMDATAERR	The pen data does not have specific pressure or height (PHW_) information. Thus, the specified PHW_ data could not be selectively trimmed. Use TPD_OEMDATA to remove all OEM information.
PDR_PNDTERR	Invalid pen data.
PDR_VERSIONERR	A version 1.0 pen data object could not be converted to the 2.0 format.

Comments

TrimPenData supplements the capabilities of **CompressPenData**. Together, these two functions replace the version 1.0 Pen API function **CompactPenData**, which is supported for compatibility only.

The data that *hpndt* points to must not be compressed. If it is, **TrimPenData** simply retrieves the original (untrimmed) pen data.

See Also

CompactPenData, **CompressPenData**, PDK_

UnhookResultsHookHREC Overview

Overview

2.0

Unhooks a recognizer result hook set with the **SetResultsHookHREC** function.

int UnhookResultsHookHREC(HREC *hrec*, **HRECHOOK** *hHook* **)**

Parameters

hrec

Module handle of the recognizer library. If this value is NULL, the system default recognizer is used.

hHook

Handle of the hook function.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter or other error.
HRCR_MEMERR	Insufficient memory.

See Also

ResultsHookHREC, **SetResultsHookHREC**

UninstallRecognizer Overview

Overview

1.0 2.0

Unloads a recognizer previously installed with **InstallRecognizer**.

void UninstallRecognizer(HREC *hrec* **)**

Parameters

hrec

Recognizer handle.

Return Value

This function does not return a value.

Comments

Windows maintains a use count for all installed recognizers and doesn't unload a recognizer until the last remaining client application has called **UninstallRecognizer**. For every call an application makes to **InstallRecognizer**, it must must make a matching call to **UninstallRecognizer**.

Before unloading a recognizer library, the system calls the recognizer's **ConfigRecognizer** function with the subfunction WCR_CLOSERECOGNIZER.

It is not necessary to uninstall the default recognizer; an application must uninstall all recognizers that it explicitly loads.

See Also

InstallRecognizer, **ConfigHREC**

UpdatePenInfo Overview

Overview

1.0 2.0

Notifies the RC Manager that a **PENINFO** value has changed. This function is called by pen drivers compatible with version 1.0 of the Pen API.

Note This function is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

void UpdatePenInfo(LPPENINFO *lppeninfo* **)**

Parameters

lppeninfo

Address of a **PENINFO** structure containing the new information.

Return Value

This function does not return a value.

Comments

A **PENINFO** value may change when the user alters the driver parameters in the configuration dialog box. When this happens, the pen driver must call **UpdatePenInfo** to notify the RC Manager of the change.

See Also

PENINFO

WriteHWL Overview

Overview

2.0

Writes a word list to a file.

int WriteHWL(HWL *hwl*, **HFILE** *hfile* **)**

Parameters

hwl

Handle to a word list.

hfile

A handle to a file previously opened for writing.

Return Value

Returns HRCR_OK if successful; otherwise, returns one of the following negative values:

Constant	Description
HRCR_ERROR	Invalid parameter, or file or other error.
HRCR_MEMERR	Insufficient memory.

Comments

The words are saved as ANSI text, one word per line, followed by a carriage return and linefeed. The file must already exist and be open for writing. An application can append to the file by positioning the file pointer at the end before calling **WriteHWL**. In this context, a word can represent a phrase and contain spaces or other noncharacters, such as "New York" and "ne're-do-well."

For a description of word lists and how a recognizer uses them, see "Configuring the HRC" in Chapter 5, "The Recognition Process."

See Also

CreateHWL, **ReadHWL**

Pen Application Programming Interface Structures

This chapter describes in alphabetical order the structures defined by the Pen Application Programming Interface (API). Each entry includes the structure **typedef** definition, descriptions of the structure members, and cross-references where appropriate. The entry heading identifies the Pen API version, such as 1.0 or 2.0, that supports the structure.

ABSTIME Overview

Overview

2.0

Absolute time structure.

```
typedef struct {
     DWORD sec;
     UINT ms;
} ABSTIME;
```

Members

sec

Number of seconds since 12:00 A.M. of January 1, 1970, as returned by the C run time library **time** function.

ms

Additional offset in milliseconds. This member can be any value from 0 through 999.

See Also

GetStrokeAttributes, SetStrokeAttributes, INTERVAL

ANIMATEINFO Overview

Overview

2.0

Animation information used by the **DrawPenDataEx** function for animation control.

```
typedef struct {
     DWORD cbSize;
     UINT uSpeedPct;
     UINT uPeriodCB;
     UINT fuFlags;
     LPARAM lParam;
     DWORD dwReserved;
} ANIMATEINFO;
```

Members

cbSize

Size of this structure in bytes.

uSpeedPct

Drawing speed, expressed as a percentage of the user's entry speed. To redraw pen data at the same speed at which it was created, this value should be set to 100. A value of 0 halts drawing. Setting **uSpeedPct** to 0 is valid only if the *lpfnAnimateCB* parameter of **DrawPenDataEx** is defined. Otherwise, the drawing halts with no way to restart it. **uSpeedPct** can be changed by a call-back function.

uPeriodCB

Callback period in milliseconds. Typical values are 1 (very fast), 250 (fast), 1000 (slow), or 0 (never). Any value in **uPeriodCB** is ignored if the **DrawPenDataEx** argument *lpfnAnimateCB* is NULL. **uPeriodCB** may also be AI_CBSTROKE, to indicate that the callback should occur after each stroke is drawn.

fuFlags

Flags that control animation (can be 0). The AI_SKIPUPSTROKES option specifies that the time taken to account for the points in the up strokes should be ignored. If this flag is 0, and if the value in **uSpeedPct** is small enough, there will be a delay between pen-down strokes, reflecting the user's inter-stroke delay during creation of the pen data.

lParam

Application value to pass to the callback function set up by the *lpfnAnimateCB* argument of **DrawPenDataEx**.

dwReserved

Must be 0.

Comments

Before using **ANIMATEINFO**, an application must initialize **cbSize** with sizeof(ANIMATEINFO).

See Also

AnimateProc, **DrawPenDataEx**

BOXEDITINFO Overview

Overview

2.0

Size information for boxed edit control.

```
typedef struct {
     int cxBox;
     int cyBox;
     int cxBase;
     int cyBase;
     int cyMid;
     BOXLAYOUT boxlayout;
     UINT wFlags;
     BYTE szFaceName[BEI_FACESIZE];
     UINT wFontHeight;
     UINT rgwReserved[8];
} BOXEDITINFO;
```

Members

cxBox

Width of a single box.

cyBox

Height of a single box.

cxBase

In-box x-margin to guideline.

cyBase

In-box y-offset from top to baseline.

cyMid

Reserved for future use; must be set to 0.

boxlayout

BOXLAYOUT structure.

wFlags

Flags specifying boxed edit options. Currently, the only defined option is BEIF_BOXCROSS.

szFaceName[BEI_FACESIZE]

Font face name, where BEI_FACESIZE is defined as 32.

wFontHeight

Font height.

rgwReserved[8]

Reserved for future use; must be set to 0.

See Also

BOXLAYOUT

BOXLAYOUT Overview

Overview

1.0 2.0

Specifies some of the characteristics of a bedit control. The **GUIDE** and **BOXEDITINFO** structures determine the rest. The HE_GETBOXLAYOUT and HE_SETBOXLAYOUT *wParam* values of the WM_PENCTL message retrieve and set the **BOXLAYOUT** structure for a bedit control.

For more details, see the WM_PENCTL message.

```
typedef struct {
     int cyCusp;
     int cyEndCusp;
     UINT style;
     DWORD dwReserved1;
     DWORD dwReserved2;
     DWORD dwReserved3;
} BOXLAYOUT;
```

Members

cyCusp

Height of the box in pixels when BXS_RECT is specified; otherwise, height of the cusp in pixels (in comb style).

cyEndCusp

Height of cusps, in pixels, at extreme ends.

style

Bitwise-OR combination of the following BXS_ flags:

Constant	Description
BXS_NONE	Default comb style.
BXS_RECT	Rectangular boxes (instead of comb style).
BXS_BOXCROSS	(Japanese version only.) Rectangular boxes with a small cross at the center of each cell. Note that any state set via this flag (or the absence of it) may be overridden by the user's selection of the BOXCROSS setting in the Bedit Control Panel.

dwReserved1

Reserved; must be set to 0.

dwReserved2

Reserved; must be set to 0.

dwReserved3

Reserved; must be set to 0.

Comments

The following table lists the default values for the **BOXLAYOUT** structure.

Value	Description
cyCusp	Equivalent in pixels of BXD_CUSPHEIGHT dialog units.
cyEndCusp	Equivalent in pixels of BXD_ENDCUSPHEIGHT dialog units.
style	Comb style.

Figure 11.1 shows the general layout of a boxed edit control. Some of the terms in the figure are explained in the reference entry for the **GUIDE** structure. Figure 11.2 shows an individual cell from a boxed edit control.

{ewc msdncd, EWGraphic, bsd23554 0 /a "SDK_1A.BMP"}

{ewc msdncd, EWGraphic, bsd23554 1 /a "SDK_1B.BMP"}

See Also

BOXEDITINFO, WM_PENCTL, **GUIDE**, BXD_

BOXRESULTS Overview

Overview

2.0

Contains box results for the **GetBoxResultsHRC** function.

```
typedef struct {
     int indxBox;
     HINKSET hinksetBox;
     SYV rgSyv[1];
} BOXRESULTS;
```

Members

indxBox

Index of the box with respect to the **GUIDE** structure.

hinksetBox

An inkset representing the pen data that belongs to the box, if requested by the **GetBoxResultsHRC** function. This member can be NULL.

rgSyv[1]

Variable-length array of alternative guesses made by the recognizer. The guesses are arranged in descending order of confidence, so that the first alternative in the array is the most likely choice.

See Also

GetBoxResultsHRC, **GUIDE**

CALBSTRUCT Overview

Overview

1.0 2.0

Pen calibration information.

```
typedef struct {
     int wOffsetX;
     int wOffsetY;
     int wDistinctWidth;
     int wDistinctHeight;
} CALBSTRUCT;
```

Members

wOffsetX

Value in tablet units to add to x-coordinates for proper calibration.

wOffsetY

Value in tablet units to add to y-coordinates for proper calibration.

wDistinctWidth

Specifies the number of distinct x-coordinates the tablet can detect.

wDistinctHeight

Specifies the number of distinct y-coordinates the tablet can detect. The **wDistinctWidth** and **wDistinctHeight** members have the same meanings and values as the identically-named members in the **PENINFO** structure.

CTLINITBEDIT Overview

Overview

2.0

Initialization information for a boxed edit (bedit) control.

```
typedef struct {
     DWORD cbSize;
     HWND hwnd;
     int id;
     int wSizeCategory;
     WORD wFlags;
     DWORD dwReserved;
} CTLINITBEDIT;
```

Members

cbSize

Size of this structure in bytes.

hwnd

Handle of a boxed edit window.

id

Control identifier.

wSizeCategory

Size category, which can be one of the following BESC_ constants:

Constant	Description
BESC_DEFAULT	Use the default size parameters to create the boxed edit control. This results in the same behavior as BESC_KANJIFIXED for applications that have registered themselves through the use of the **SetPenAppFlags** function with the RPI_KANJIFIXEDBEDIT flag. For all other applications, it results in the same behavior as BESC_ROMANFIXED.
BESC_ROMANFIXED	Comb-style bedit control with dimensions indicated by BXD_ constants (in dialog units). Meant for use with Roman characters.
BESC_KANJIFIXED	(Japanese version only.) Box-style bedit control with dimensions indicated by BXDK_ constants (in dialog units). Meant for use with kanji characters. This value should be used by applications that cannot handle user-defined box sizes.
BESC_USERDEFINED	A bedit control that can handle the box

size parameters defined by the user. For further details, see the description for PMSC_BEDITCHANGE in the reference section for the WM_PENMISC message.

wFlags

Flags that determine certain properties of the boxed edit control. This can be a combination of the following values:

Constant	Description
CIB_NOGDMSG	(Not supported in Japanese version.) Do not display the garbage-detection message box when writing in the bedit control.
CIB_NOACTIONHANDLE	Do not create action handles.
CIB_NOFLASHCURSOR	Do not change the cursor if tap-and-hold action is detected.
CIB_NOWRITING	(Japanese version only.) Do not allow pen input into the control. Other methods of inputting text, such as keyboard input or pasting from the keyboard, are allowed.

dwReserved

Reserved, should be set to 0.

Comments

Before using **CTLINITBEDIT**, an application must initialize **cbSize** with sizeof(CTLINITBEDIT).

See Also

WM_CTLINIT

CTLINITHEDIT Overview

Overview

2.0

Initialization information for a handwriting edit (hedit) control.

```
typedef struct {
     DWORD cbSize;
     HWND hwnd;
     int id;
     DWORD dwFlags;
     DWORD dwReserved;
} CTLINITHEDIT;
```

Members

cbSize

Size of this structure in bytes.

hwnd

Handle of boxed edit window.

id

Control identifier.

dwFlags

Flags that determine some properties of the hedit control. This can be a combination of the following values:

Constant	Description
CIH_NOGDMSG	(Not supported in Japanese version.) Do not put up the garbage-detection message box when writing in this hedit control.
CIH_NOACTIONHANDLE	Do not create action handles for this hedit control.
CIH_NOEDITTEXT	Do not show the edit text, insert text, or writing tool dialogs when writing in this hedit control.
CIH_NOFLASHCURSOR	Do not change the cursor while doing tap-and-hold selection in this hedit control.

dwReserved

Reserved, should be set to 0.

Comments

Before using **CTLINITHEDIT**, an application must initialize **cbSize** with sizeof(CTLINITHEDIT).

See Also

WM_CTLINIT

CTLINITIEDIT Overview

Overview

2.0

Specifies the initial settings and options of an ink edit (iedit) control. A pointer to this structure is passed to the parent window of the control as the *lParam* parameter of the WM_CTLINIT message. This forms the last step of the control's processing of the WM_CREATE message.

```
typedef struct {
     DWORD cbSize;
     HWND hwnd;
     int id;
     WORD ieb;
     WORD iedo;
     WORD iei;
     WORD ien;
     WORD ierec;
     WORD ies;
     WORD iesec;
     HPENDATA hpndt;
     WORD pdts;
     HGDIOBJ hgdiobj;
     HPEN hpenGrid;
     POINT ptOrgGrid;
     WORD wVGrid;
     WORD wHGrid;
     DWORD dwApp;
     DWORD dwReserved;
} CTLINITIEDIT;
```

Members

cbSize

Size of this structure in bytes.

hwnd

Handle to an ink edit window.

id

Control identifier.

ieb

Background IEB_ bit values (see IE_SETBKGND).

iedo

Draw options IEDO_ bit values (see IE_SETDRAWOPTS).

iei

Ink input IEI_ bit values (see IE_SETINKINPUT).

ien

Notification IEN_ bit values (see IE_SETNOTIFY).

ierec

Recognition IEREC_ bit values (see IE_SETRECOG).

ies

Style IES_ bit values (see IE_GETSTYLE).

iesec

Security IESEC_ bit values (see IE_SETSECURITY).

hpndt

Initial pen data.

pdts

Initial map mode.

hgdiobj

Brush or bitmap, depending on background bits option in **ieb**.

hpenGrid

Pen to use in drawing grid.

ptOrgGrid

Point of origin for the grid lines.

wVGrid

Vertical grid line spacing.

wHGrid

Horizontal grid line spacing.

dwApp

Application data.

dwReserved

Reserved.

Comments

Before using **CTLINITIEDIT**, an application must initialize **cbSize** with sizeof(CTLINITIEDIT).

See Also

IE_SETBKGND, IE_SETDRAWOPTS, IE_SETNOTIFY, IE_SETRECOG, IE_SETSECURITY, IE_GETSTYLE, IE_SETDRAWOPTS, WM_CTLINIT

CWX Overview

Overview

2.0

Specifies optional parameters for the **CorrectWritingEx** function. (Japanese version only.)

```
typedef struct {
     DWORD cbSize;
     WORD wApplyFlags;
     HWND hwndText;
     HRC hrc;
     char szCaption[CBCAPTIONCWX];
     DWORD dwEditStyle;
     DWORD dwSel;
     DWORD dwFlags;
     WORD ixkb;
     WORD rgState[CKBCWX];
     POINT ptUL;
     SIZE sizeHW;
} CWX;
```

Members

cbSize

Size of this structure in bytes. This field must be initialized to sizeof(CWX).

wApplyFlags

Options to specify which members of this structure are to be used to override the most-recently-used values provided by default; not all fields qualify. If this value is 0, the most-recently-used settings will be shown; otherwise, this value can be a combination of the following values:

Constant	Description
CWXA_CONTEXT	Use the *dwFlags* member to specify context.
CWXA_KBD	Use the *ixkb* member to specify a keyboard.
CWXA_STATE	Apply the states provided in the *rgState* array.
CWXA_PTUL	Move the dialog box upper corner to the screen position specified by the *ptUL* member.
CWXA_SIZE	Use the window size specified by the *sizeHW* member while using the handwriting recognition tab.
CWXA_NOUPDATEMRU	Do not update the registry with the last state of the correction dialog. This causes any changes made to the position and state of the Data Input Window to be discarded. This has no effect on user changes to the text, however.

hwndText

Text window to which to send WM_GETTEXT and WM_SETTEXT messages. If this is NULL, the owner of the Correct Writing dialog box will be used.

hrc

Handle to a recognition context. If this is NULL, a WM_PENMISC message with the wParam parameter of PMSC_GETHRC will be sent to the owner window to get a recognition context. If that too is NULL, then a default context will be used. The system will destroy its copy of *hrc* before the call returns.

szCaption[CBCAPTIONCWX]

A null-terminated array of characters to be used for a dialog caption. If this string has 0 length, then the default caption "Edit Text" will be used. The maximum length of caption allowed is specified by the CBCAPTIONCWX constant.

dwEditStyle

Style to use for the Data Input Window's edit control. By default this is ES_LEFT. If this style includes ES_MULTILINE, entry of Return and Tab characters is allowed; otherwise, they are not allowed. In any case, the style of the actual edit control will look like a multiline edit control.

dwSel

Specifies the selection. The low-order word (LOWORD) is the start position and the high-order word (HIWORD) is the end position. The default values are 0 for start and 0xFFFF for end, to select all text.

dwFlags

Specifies context flags, provided that the CWXA_CONTEXT bit is set in the **wApplyFlags** member; otherwise, the most-recently-used context flags are used and this field is ignored. On return, this field contains the updated flags. The flags may be CWX_DEFAULT (0), or a bitwise-OR combination of the following constant values:

Constant	Description
CWX_TOPMOST	Specifies that the dialog window is to be a topmost window. The window is not topmost by default.
CWX_NOTOOLTIPS	Disables showing tool tips for graphical buttons. They are shown by default.
CWX_JPERIOD	Specifies that the Japanese period is to be used on some keys on the Data Input Window keypads. The English period is used by default.
CWX_JCOMMA	Specifies that the Japanese comma is to be used on some keys on the Data Input Window keypads. The English comma is used by default.
CWX_DEFAULT	Zero; none of the above flags are set.

ixkb

Specifies which Data Input Window keyboard, or handwriting input, is to be used first, provided that the CWXA_KBD bit is set in the **wApplyFlags** member; otherwise, the most-recently use keyboard is used and this field is ignored. On return, this field contains the updated keyboard identifier. This may be one of the following values:

Constant	Description

CWXK_HW	Handwriting, not keyboard, input. If this value is specified, most of the dialog will be available for handwriting input, and the dialog will be sizable.
CWXK_50	50-On keyboard.
CWXK_QWERTY	QWERTY keyboard, including Hiragana, Katakana, and Romaji-to-Kana conversion alternative states.
CWXK_ROMAJI	Condensed Romaji-to-Kana keyboard, similar to some pocket computers.
CWXK_NUM	Numeric and Telephone keyboard.
CWXK_KANJI	Kanji keyboard, which provides a method of specifying a Kanji character based on its strokes.
CWXK_CODE	Kanji Code Finder keyboard, which allows the lookup of a Kanji character based on its JIS, Shift-JIS, or Kuten code value.
CWXK_YOMI	Kanji character finder based on the sound, or "reading" (Yomi) of the character.

rgState[CKBCWX]

An array of keyboard states with which to initialize the CKBCWX number of keyboards, provided that the CWXA_STATE bit is set in the **wApplyFlags** member; otherwise, the most-recently-used states are used and this member is ignored. On return, this member contains the updated states. Each element of the array may be CWXKS_DEFAULT (0, which is equivalent to CWXKS_HAN + CWXKS_ROMA), or a bitwise-OR combination of the following constants:

Constant	Description
CWXKS_CAPS	Set CAPSLOCK state on QWERTY keyboard.
CWXKS_HAN	Set Hankaku (single-byte) state.
CWXKS_ZEN	Set Zenkaku (double-byte) state.
CWXKS_ROMA	Set Romaji characters state.
CWXKS_HIRA	Set Hiragana characters state.
CWXKS_KATA	Set Katakana characters state.

ptUL

Specifies the upper-left corner of the dialog in screen coordinates, provided that the CWXA_PTUL bit is set in the **wApplyFlags** member; otherwise, the most-recently-used position is used and this member is ignored. On return, this member contains the updated screen position of the upper-left corner.

sizeHW

Specifies the size of the dialog when it is in handwriting input mode, provided that the CWXA_SIZE bit is set in the **wApplyFlags** member; otherwise, the most-recently-used size is used and this field is ignored. On return, this field contains the updated size.

Comments

Note that even if some bits are not set in **wApplyFlags**, the corresponding structure members are still updated with the last-used values on return.

See Also

CorrectWritingEx

GUIDE Overview

Overview

1.0 2.0

Specifies the characteristics of any guidelines used in the writing area.

```
typedef struct {
     int xOrigin;
     int yOrigin;
     int cxBox;
     int cyBox;
     int cxBase;
     int cyBase;
     int cHorzBox;
     int cVertBox;
     int cyMid;
} GUIDE;
```

Members

xOrigin

Position of left edge of the first box in screen coordinates.

yOrigin

Position of top edge of the first box in screen coordinates.

cxBox

Width of each box in screen pixels.

cyBox

Height of each box in screen pixels.

cxBase

Margin to the guideline. This is one-half the distance in pixels between adjacent boxes.

cyBase

Vertical distance in pixels from the baseline to the top of the box.

cHorzBox

Number of columns of boxes.

cVertBox

Number of rows of boxes.

cyMid

Distance in pixels from the baseline to the midline, or 0 if midline is not present.

Comments

If the application has drawn guidelines on the screen on which the user is expected to write, the application should set the values in the **GUIDE** structure to inform the recognizer. The **GUIDE** structure is for the recognizer's use only. Setting the **GUIDE** structure does not by itself draw any visual clues on the display. It is the responsibility of the application or the control to draw the visual clues. The appearance of

a boxed edit control is determined by the **BOXLAYOUT** and **GUIDE** structures together.

The **xOrigin** and **yOrigin** members are screen coordinates of the top-left corner of the area to write in. The **cyBox** and **cxBox** members are the height and width of the individual boxes to write in. The **cHorzBox** and **cVertBox** members specify the number of columns and rows. **cyBase** specifies a baseline within the box. (Setting **cyBase** to 0 indicates no baseline.) The **cxBase** member gives a horizontal displacement of the edge of the guideline from the edge of the box where writing is expected to start.

If only horizontal lines are present, set **cxBox** to 0. In this case, only **yOrigin**, **cyBox**, **cyBase**, and **cyMid** are valid. A default **GUIDE** structure has all elements set to 0.

To establish a guide, initialize a **GUIDE** structure and set it into an **HRC** with the **SetGuideHRC** function. This also applies to a standard bedit, as demonstrated in "The bedit Control" in Chapter 3.

For boxed input, the **GetBoxMappingHRCRESULT** function returns an index to the box containing the requested input character. This is numbered in zero-based row-major order. In Figure 11.3 below, for example, the "h" character is in box 12.

{ewc msdncd, EWGraphic, bsd23554 2 /a "SDK_2.BMP"}

{ewc msdncd, EWGraphic, bsd23554 3 /a "SDK_3.BMP"}

For best recognition results, the pair-wise ratios of **cxBox**, **cyBox**, and **cyBase** should be similar to the default ratios.

See Also

SetGuideHRC, **BOXLAYOUT**, BXD_

INKINGINFO Overview

Overview

2.0

Provides information about where and how the system should display ink.

```
typedef struct {
     DWORD cbSize;
     UINT wFlags;
     PENTIP tip;
     RECT rectClip;
     RECT rectInkStop;
     HRGN hrgnClip;
     HRGN hrgnInkStop;
} INKINGINFO;
```

Members

cbSize

Size of this structure in bytes.

wFlags

A bitwise-OR combination of the following PII_ flags:

Constant	Description
PII_INKPENTIP	Use **tip** for pen characteristics.
PII_INKCLIPRECT	Clip ink using **rectClip**.
PII_INKSTOPRECT	Terminate inking on a pen-down event inside **rectInkStop**.
PII_INKCLIPRGN	Clip ink using **hrgnClip**. If **hrgnClip** is set, any value in **rectClip** is disregarded.
PII_INKSTOPRGN	Terminate inking on a pen-down event inside **hrgnInkStop**.
PII_SAVEBACKGROUND	Save the background that is being inked on. The saved background is restored when the current input session terminates.
PII_CLIPSTOP	Directs Windows to stop inking if the pen goes down outside **rectClip** or **hrgnClip**, if either have been set.

tip

A **PENTIP** structure defining the pen type, size, and color.

rectClip

Clipping rectangle for the ink. Setting **rectClip** to {-32767, -32767, 32767, 32767} is equivalent to having no clipping region.

rectInkStop

Rectangle in which a pen-down event stops inking. Setting **rectInkStop** to empty is equivalent to not having an ink stop region.

hrgnClip

Clipping region for the ink. Setting **hrgnClip** to NULL is equivalent to not having a clipping region.

hrgnInkStop

Region in which a pen-down event stops inking. Setting **hrgnInkStop** to NULL is equivalent to not having an ink stop region.

Comments

All areas are in screen coordinates.

The **wFlags** member specifies which of the other members contain valid information. For example, if PII_INKCLIPRECT is set in **wFlags**, the **rectClip** member specifies the clipping rectangle. Otherwise, a default value is used.

Before using **INKINGINFO**, an application must initialize **cbSize** with sizeof(INKINGINFO).

See Also

PENTIP, **StartInking**, WM_PENEVENT

INPPARAMS Overview

Overview

2.0

Describes a set of targets.

```
typedef struct {
     DWORD cbSize;
     DWORD dwFlags;
     HPENDATA hpndt;
     TARGET target;
} INPPARAMS;
```

Members

cbSize

Size of this structure in bytes.

dwFlags

Reserved for future use; must be 0.

hpndt

Handle to a pen data object.

target

A **TARGET** structure where input is directed.

Comments

Before using **INPPARAMS**, an application must initialize **cbSize** with sizeof(INPPARAMS).

See Also

TARGET

INTERVAL Overview

Overview

2.0

Interval structure for inksets.

```
typedef struct {
     ABSTIME atBegin;
     ABSTIME atEnd;
} INTERVAL;
```

Members

atBegin

Beginning of 1-millisecond granularity interval.

atEnd

Time at 1 millisecond past end of interval.

See Also

ABSTIME

OEMPENINFO Overview

Overview

1.0 2.0

Structure containing original equipment manufacturer (OEM) hardware information for the pen or tablet.

```
typedef struct {
     UINT wPdt;
     UINT wValueMax;
     UINT wDistinct;
} OEMPENINFO;
```

Members

wPdt

A combination of PDT_ values.

wValueMax

The largest value returned by the device.

wDistinct

The number of distinct readings possible.

Comments

The **OEMPENINFO** structure contains a description of the additional OEM information that the hardware can generate. It is a component of the **PENINFO** structure.

Besides capturing the x- and y- coordinates of the pen movement, a pen device has the option of providing a number of other types of input data, such as pen pressure, height of the pen tip above the tablet surface, angle of the pen, and so on. A pen driver can capture up to MAXOEMDATAWORDS types of data, where MAXOEMDATAWORDS is defined as six. An application can access the OEM data through the **GetPenInput** function. A recognizer can receive OEM data from an application through the **AddPenInputHRC** function. It is the up to the application whether to send this data or not.

Each pen event generates a packet of information from the pen driver that contains the current pen position and, optionally, other types of OEM information. The **cbOemData** member of the **PENINFO** structure specifies the width of the optional OEM data in bytes. Each type of data is one word wide. The type of data in the *n*th word of the OEM data packet is given by the *n*th element of the **rgoempeninfo** member (an array of **OEMPENINFO** structures) in the **PENINFO** structure.

For the **wPdt** member, PDT_NULL indicates no data. Values greater than PDT_OEMSPECIFIC are reserved for private use by drivers for data types not currently defined as standard. The **wValueMax** member contains the largest variable size the device can return for that data type. The **wDistinct** member is the number of distinct readings the device can take between 0 and **wValueMax**.

For a list of values for the **wPdt** member, see the entry for PDT_ values in Chapter 13, "Pen Application Programming Interface Constants."

Example

As an example of how to use **OEMPENINFO**, consider a device that can sense both the height above the tablet surface and the Z-angle of the pen. Assume the device can sense 256 levels of height in a range from 0 to 10 centimeters and has a resolution of 1 degree on the angle of the pen. The two additional words of OEM information occupy 4 bytes, so the **cbOemData** and **rgoempeninfo** members of **PENINFO** look like this:

```
peninfo.cbOemData = 4
peninfo.rgoempeninfo[MAXOEMDATAWORDS] = {
                                                    {PDT_HEIGHT,
1000, 256},
                                                    {PDT_ANGLEZ,
1800, 180},
                                                    {PDT_NULL, 0, 0},
                                                    {PDT_NULL, 0, 0},
                                                    {PDT_NULL, 0, 0},
                                                    {PDT_NULL, 0,
0} };
```

This optional information is saved by the pen driver in the same manner as the x- and y- coordinate data. There must be a one-to-one correspondence between the OEM event data and the coordinate data.

Figure 11.5 shows the pen in a position where both the Xy-angle and Z-angle are approximately 45 degrees.

{ewc msdncd, EWGraphic, bsd23554 4 /a "SDK_4.BMP"}

See Also

PENINFO

OEM_PENPACKET

2.0

A pen packet used by Pen API, version 2.0, consisting of the information received from the pen device for a single sample. For a definition of pen packet, see **SetPenHook**.

```
typedef struct {
     UINT wTabletX;
     UINT wTabletY;
     UINT wPDK;
     UINT rgwOemData[MAXOEMDATAWORDS];
     DWORD dwTime;
} PENPACKET;
```

Members

wTabletX

The x-dimension in raw tablet coordinates.

wTabletY

The y-dimension in raw tablet coordinates.

wPDK

Pen hardware state bits, expressed as a combination of PDK_ values.

rgwOemData[MAXOEMDATAWORDS]

Array of OEM-specific data. MAXOEMDATAWORDS is defined as 6.

dwTime

Time stamp indicating when the pen packet originated.

Comments

A pen packet is the basic unit of communication between the pen driver and Windows. A pen packet contains all of the information about a single logical event: x-y coordinate position, button states, and any optional information such as pressure or barrel rotation. Several physical events—that is, interrupts—may be needed to construct a single logical event.

The **rgwOemData** member contains the data relating to the OEM hardware, such as pen pressure, angle, and so forth.

See Also

SetPenHookCallback, **OEMPENINFO**, **PENPACKET**

PCMINFO Overview

Overview

2.0

Pen collection mode information. All regions and rectangles are in screen coordinates. Time-out values are in milliseconds.

```
typedef struct {
     DWORD cbSize;
     DWORD dwPcm;
     RECT rectBound;
     RECT rectExclude;
     HRGN hrgnBound;
     HRGN hrgnExclude;
     DWORD dwTimeout;
} PCMINFO;
```

Members

cbSize

Size of this structure in bytes.

dwPcm

A combination of PCM_ flags specifying options for pen collection.

rectBound

Bounding rectangle for pen collection.

rectExclude

Exclusion rectangle for pen collection.

hrgnBound

Bounding region for pen collection.

hrgnExclude

Exclusion region for pen collection.

dwTimeout

Time-out before terminating pen collection.

Comments

Before using **PCMINFO**, an application must initialize **cbSize** with sizeof(PCMINFO).

See Also

StartPenInput, WM_PENEVENT, PCM_

PDEVENT Overview

Overview

2.0

Provides details of the pointing-device event that is the subject of an IN_PDEVENT notification. A pointing-device event can be a pen tap, mouse double-click, and so on. This structure is returned by the IE_GETPDEVENT message.

```
typedef struct {
     DWORD cbSize;
     HWND hwnd;
     UINT wm;
     WPARAM wParam;
     LPARAM lParam;
     POINT pt;
     BOOL fPen;
     LONG lExInfo;
     DWORD dwReserved;
} PDEVENT;
```

Members

cbSize

Size of this structure in bytes.

hwnd

Handle to ink edit window.

wm

Window WM_ message.

wParam

wParam of event.

lParam

lParam of event.

pt

Event point in ink edit client coordinates.

fPen

TRUE if pen event, FALSE if mouse event.

lExInfo

Windows **GetMessageExtraInfo** function return value.

dwReserved

Reserved.

Comments

Before using **PDEVENT**, an application must initialize **cbSize** with sizeof(PDEVENT).

For descriptions of the WM_ messages that pertain to pen-based computing, refer to Chapter 12, "Pen Application Programming Interface Messages."

See Also

IE_GETPDEVENT, IN_PDEVENT

PENDATAHEADER Overview

Overview

1.0 2.0

Main header of an **HPENDATA** memory block.

```
typedef struct {
     UINT wVersion;
     UINT cbSizeUsed;
     UINT cStrokes;
     UINT cPnt;
     UINT cPntStrokeMax;
     RECT rectBound;
     UINT wPndts;
     int nInkWidth;
     DWORD rgbInk;
} PENDATAHEADER;
```

Members

wVersion

Pen data format version. Same as the version number for the Pen API, which is currently 0x0002. Calling **GetPenDataAttributes** with the GPA_VERSION argument retrieves the value of **wVersion**.

cbSizeUsed

Size (in bytes) of pen data memory block.

cStrokes

Number of strokes in the block. (Each pen-down and pen-up sequence counts as a single stroke.)

cPnt

Count of all points in the block. Calling **GetPenDataAttributes** with the GPA_POINTS argument retrieves the value of **cPnt**.

cPntStrokeMax

Length (in points) of longest stroke. Calling **GetPenDataAttributes** with the GPA_MAXLEN argument retrieves the value of **cPntStrokeMax**.

rectBound

Bounding rectangle of all pen-down points.

wPndts

Data scaling metric value, expressed as a bitwise-OR combination of PDTS_ values.

nInkWidth

Ink width, in pixels.

rgbInk

Ink color.

Comments

The **PENDATAHEADER** structure describes the contents of an **HPENDATA** memory block. Use the

GetPenDataInfo or **GetPenDataAttributes** function to retrieve information from a **PENDATAHEADER** structure.

For a description of the **HPENDATA** memory block, see "The HPENDATA Object" in Chapter 4, "The Inking Process."

For a list of data scaling values, refer to the entry for PDTS_ values in Chapter 13, "Pen Application Programming Interface Constants."

See Also

GetPenDataInfo, **GetPenDataAttributes**, PDTS_

PENINFO Overview

Overview

1.0 2.0

Contains dimensions, sampling rate, and other information about the pen or tablet hardware.

```
typedef struct {
     UINT cxRawWidth;
     UINT cyRawHeight;
     UINT wDistinctWidth;
     UINT wDistinctHeight;
     int nSamplingRate;
     int nSamplingDist;
     LONG lPdc;
     int cPens;
     int cbOemData;
     OEMPENINFO rgoempeninfo[MAXOEMDATAWORDS];
     UINT rgwReserved[7];
     UINT fuOEM;
} PENINFO;
```

Members

cxRawWidth

Width of tablet in thousandths of an inch. Also specifies the maximum tablet x-coordinate.

cyRawHeight

Height of tablet in thousandths of an inch. Also specifies the maximum tablet y-coordinate.

wDistinctWidth

Number of distinct x-coordinates the hardware can detect.

wDistinctHeight

Number of distinct y-coordinates the hardware can detect. Together, the **wDistinctWidth** and **wDistinctHeight** members express the x-y resolution of the tablet. For example, if a tablet is 8 inches wide and has a resolution of 1/500 of an inch, **cxRawWidth** is 8000 and **wDistinctWidth** is 4000 because the tablet hardware can return 4000 distinct x-order values ranging from 0 to 8000.

nSamplingRate

Specifies the number of samples per second the tablet can return. This value may be less than the number of hardware interrupts per second the tablet generates because several interrupts may be required to create one pen packet sample. See the "Comments" section below for information on adjusting the sampling rate.

nSamplingDist

Specifies the distance in distinct tablet units a pen must travel before a new pen event is generated. See the "Comments" section below for information on adjusting the sampling distance.

lPdc

Pen-device capabilities, expressed as a bitwise-OR combination of the PDC_ flags.

cPens

Number of pens the tablet can simultaneously support.

cbOemData

Specifies the width, in bytes, of the additional OEM data passed in each pen packet. For example, if a tablet can detect pressure and Z-angle information, this information occupies two additional words of OEM data, so **cbOemData** is 4.

rgoempeninfo[MAXOEMDATAWORDS]

An array of **OEMPENINFO** structures. Each structure describes one word of additional OEM data contained in each pen packet. (MAXOEMDATAWORDS is defined as 6.)

rgwReserved[7]

Reserved for internal use.

fuOEM

Flags representing which OEM data to report in **rgoempeninfo**; used by applications to determine the OEM data used in an **HPENDATA** object. This member is set and used internally by the pen services and should never be modified by an application. This member is a bitwise-OR combination of the following values:

Constant	Description
PHW_ALL	Report all available OEM data.
PHW_PRESSURE	Report pressure if available.
PHW_HEIGHT	Report height if available.
PHW_ANGLEXY	Report Xy-angle if available.
PHW_ANGLEZ	Report Z-angle if available.
PHW_BARRELROTATION	Report barrel rotation if available.
PHW_OEMSPECIFIC	Report OEM-specific value if available.
PHW_PDK	Report per-point PDK_ values.

Comments

The DRV_GetPenInfo pen driver message fills a **PENINFO** structure with the current device parameters. DRV_GetPenInfo returns FALSE if a tablet is not present. If this occurs, the **PENINFO** structure that the message's *lParam1* points to is not valid. Note that an application should retrieve tablet information with the **GetPenDataInfo** and **GetPenDataAttributes** functions, rather than accessing the pen driver directly.

An application can also adjust the tablet sampling rate and sampling distance by sending the DRV_SetSamplingRate or DRV_SetSamplingDist messages to the device driver. For more information, see Appendix E, "Accessing the Pen Device Driver."

For a list of state bits for the pen driver, refer to the entry for PDK_ values in Chapter 13, "Pen Application Programming Interface Constants."

See Also

CreatePenDataEx, **GetPenDataInfo**, **TrimPenData**, **UpdatePenInfo**, **OEMPENINFO**

PENPACKET Overview

Overview

1.0 2.0

A pen packet used by Pen Windows, version 1.0, consisting of the information received from the pen device for a single sample. For a definition of pen packet, see **SetPenHook**.

```
typedef struct {
     UINT wTabletX;
     UINT wTabletY;
     UINT wPDK;
     UINT rgwOemData[MAXOEMDATAWORDS];
} PENPACKET;
```

Members

wTabletX

The x-dimension in raw tablet coordinates.

wTabletY

The y-dimension in raw tablet coordinates.

wPDK

Pen hardware state bits, expressed as a combination of PDK_ values.

rgwOemData[MAXOEMDATAWORDS]

Array of OEM-specific data. MAXOEMDATAWORDS is defined as 6.

Comments

A pen packet is the basic unit of communication between the pen driver and Windows. A pen packet contains all of the information about a single logical event: x-y coordinate position, button states, and any optional information such as pressure or barrel rotation. Several physical events—that is, interrupts—may be needed to construct a single logical event.

The **rgwOemData** member contains the real-time values associated with the pen data types described in the entry for the **OEMPENINFO** structure.

See Also

SetPenHookCallback, **OEMPENINFO**, **OEM_PENPACKET**

PENTIP Overview

Overview

2.0

Pen tip characteristics.

```
typedef struct {
     DWORD cbSize;
     BYTE btype;
     BYTE bwidth;
     BYTE bheight;
     BYTE bOpacity;
     COLORREF rgb;
     DWORD dwFlags;
     DWORD dwReserved;
} PENTIP;
```

Members

cbSize

Size of this structure in bytes.

btype

Pen nib type. Types in the range 0 through 63 are reserved for predefined standard types. An application can use values in the range 64 through 255.

bwidth

Pen nib width, in display device units (pixels).

bheight

Pen nib height, in display device units (pixels). In the current version of the Pen API, this member is ignored.

bOpacity

Opacity of the ink, which corresponds to the JOT standard. **bOpacity** must have one of the following values:

Constant	Description
PENTIP_OPAQUE	New ink overwrites any existing ink.
PENTIP_HILITE	New ink is visible but partly transparent, possibly interacting with underlying ink.
PENTIP_TRANSPARENT	Ink is completely transparent. There is no interaction with any underlying ink.

rgb

RGB pen color.

dwFlags

Reserved.

dwReserved

Reserved; must be set to 0.

Comments

Before using **PENTIP**, an application must initialize **cbSize** with sizeof(PENTIP).

See Also

GetStrokeAttributes, **SetStrokeAttributes**, **GetStrokeTableAttributes**, **SetStrokeTableAttributes**, **GetPenMiscInfo**, **SetPenMiscInfo**, **INKINGINFO**

RC Overview

Overview

1.0 2.0

Defines a recognition context (RC) for applications compatible with version 1.0 of the Pen API. Applications that do not call the superseded functions **Recognize** or **RecognizeData** do not use the **RC** structure. These applications instead use **HRC** objects, which render **RC** obsolete.

The **RC** structure is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

```
typedef struct {
     HREC hrec;
     HWND hwnd;
     UINT wEventRef;
     UINT wRcPreferences;
     LONG lRcOptions;
     RCYIELDPROC lpfnYield;
     BYTE lpUser[cbRcUserMax];
     UINT wCountry;
     UINT wIntlPreferences;
     char lpLanguage[cbRcLanguageMax];
     LPDF rglpdf[MAXDICTIONARIES];
     UINT wTryDictionary;
     CL clErrorLevel;
     ALC alc;
     ALC alcPriority;
     BYTE rgbfAlc[cbRcrgbfAlcMax];
     UINT wResultMode;
     UINT wTimeOut;
     LONG lPcm;
     RECT rectBound;
     RECT rectExclude;
     GUIDE guide;
     UINT wRcOrient;
     UINT wRcDirect;
     int nInkWidth;
     COLORREF rgbInk;
     DWORD dwAppParam;
     DWORD dwDictParam;
     DWORD dwRecognizer;
     UINT rgwReserved[cwRcReservedMax];
} RC;
```

Members

hrec

Handle of recognizer to use.

hwnd

Window to which results are sent.

wEventRef

Index into ink buffer.

wRcPreferences

Flags specifying preferences, described in "Comments" section.

lRcOptions

Recognition options, described in "Comments" section.

lpfnYield

Procedure called during processing of the **Yield** Windows function.

lpUser[cbRcUserMax]

Current writer.

wCountry

Country code.

wIntlPreferences

Flags for international preferences.

lpLanguage[cbRcLanguageMax]

Language strings.

rglpdf[MAXDICTIONARIES]

List of dictionary functions.

wTryDictionary

Maximum enumerations to search.

clErrorLevel

Level where recognizer should reject input.

alc

Enabled ALC_ alphabet codes.

alcPriority

Sets priority of the ALC_ codes.

rgbfAlc[cbRcrgbfAlcMax]

Bit field for enabled characters.

wResultMode

Result return mode specifying when to send (either as soon as possible or when complete). The RRM_ codes are described in "Comments" section.

wTimeOut

Recognition time-out in milliseconds.

lPcm

Bitwise-OR combination of PCM_ flags for ending the recognition session.

rectBound

Bounding rectangle for inking. By default, the rectangle is in screen coordinates.

rectExclude

A pen-down event inside this rectangle terminates recognition.

guide

GUIDE structure that defines guidelines for recognizer.

wRcOrient

Orientation of writing with regard to the tablet.

wRcDirect

Direction of writing.

nInkWidth

Ink width of 1-15 pixels. A value of 0 prevents display of the ink.

rgbInk

Ink color.

dwAppParam

For application use.

dwDictParam

For application use; to be passed on to dictionaries.

dwRecognizer

For application use; to be passed on to recognizer.

rgwReserved[cwRcReservedMax]

Reserved.

Comments

The following paragraphs discuss the **RC** members, listed in the order in which they appear in the preceding structure.

hrec

The **hrec** member is the handle of the recognizer to use. This value should be set to the value returned by a previous call to **InstallRecognizer**, or to RC_WDEFAULT for the default recognizer.

If **hrec** is NULL, no recognizer is used. WM_RCRESULT messages are generated as with a real recognizer, but the **wResultsType** member of **RCRESULT** is set to RCRT_NORECOG, and the **hSyv** and **lpsyv** members are set to NULL. (For a list of other values in **wResultsType**, see the entry for RCRT_ values in Chapter 13, "Pen Application Programming Interface Constants.")

hwnd

The **hwnd** member specifies the window to send recognition results to. This member cannot be NULL. Also, the mouse capture is set to this window to clear the queue of pending mouse messages that were meant for recognition.

wEventRef

The value for **wEventRef** indicates which tablet data to begin recognition with. The **wEventRef** member is returned from the **GetMessageExtraInfo** function.

InitRC sets this member to RC_WDEFAULT. If **Recognize** is called during the processing of the WM_LBUTTONDOWN message that initiates the input session, the application need take no other action.

Before an application starts recognition on some other Windows event, it should use **GetMessageExtraInfo** to save the event reference of the appropriate mouse message and place this value in **wEventRef** before calling **Recognize**.

This member is not used on calls to **RecognizeData**.

wRcPreferences

The **wRcPreferences** member specifies the user preferences as a combination of RCP_ constants.

lRcOptions

The **lRcOptions** member specifies various options for recognition. It is a bitwise-OR combination of RCO_ constants.

lpfnYield

The **lpfnYield** member points to a callback function used by the recognizer before it yields. The application sets this to NULL for no yield processing. Recognition can often take more than a few seconds; therefore, a recognizer should periodically call the yield function to yield control to other Windows tasks. The default yield function is:

```
BOOL FAR PASCAL StandardYieldFunction()
{
     Yield( );
     return 1;
}
```

If **Recognize** or **RecognizeData** is called with **lpfnYield** set to RC_LDEFAULT, then the default yield function is called. If the **lpfnYield** member is not NULL, the recognizer calls **lpfnYield** every time before it yields.

lpUser

```
#define    cbRcUserMax             32
BYTE lpUser[cbRcUserMax];
```

The **lpUser** member specifies the name of the current writer. The current writer is used to specify any custom prototype sets that might be available to the recognizer. If the **lpUser** member is NULL, it means that the recognizer should use the standard prototype set—that is, the prototype set as it existed before it was modified (through training, for example).

wCountry

The **wCountry** member contains the country code. The values for country code are the same as the values used by the International item of the Control Panel for the **iCountry** member in the [intl] section of the WIN.INI file.

wIntlPreferences

The **wIntlPreferences** member contains a combination of various RCIP_ flags. Currently, this member can be only 0 or RCIP_ALLANSICHAR. If 0, only characters from the current language or languages are enabled. If **wIntlPreferences** is RCIP_ALLANSICHAR, the entire ANSI character set is enabled.

lpLanguage

```
#define    cbRcLanguageMax                44
char lpLanguage[cbRcLanguageMax]
```

The **lpLanguage** member is a list of language strings. Each string is null-terminated and the list ends with a null string.

The set of values for each language string is the same as the set used by the International item of the Control Panel for the **sLanguage** member in the [Intl] section of the WIN.INI file. These three-letter codes are documented in the Microsoft Windows Software Development Kit.

A recognizer should implement recognition of the ANSI character set and then use this information during recognition to limit a match to the appropriate subset. The **lpLanguage** member holds strictly optional information; a recognizer may choose to ignore it. By definition, the character set implied by a language string is the set of characters that can be generated from the country-specific keyboard without using the ALT+numeric keypad combinations. It is still possible to enter ANSI characters outside the given language through the use of the onscreen keyboard and ALT+numeric keypad combinations.

rglpdf

```
#define    MAXDICTIONARIES 16
LPDF rglpdf[MAXDICTIONARIES]
```

The dictionary path member **rglpdf** specifies which dictionaries are called by the RC Manager to convert symbol graphs into strings.

If **rglpdf[0]** is NULL, the NULL dictionary path is used. The NULL dictionary path indicates that the first enumeration from the symbol graph is used as the best enumeration. The array of dictionary functions is null-terminated. During recognition, the dictionary functions are called in the order in which they appear. For more details, see the entry for the **DictionarySearch** function.

wTryDictionary

The **wTryDictionary** member specifies the maximum number of enumerations generated from the symbol graph during dictionary processing on the results of recognition. The minimum number allowed is 1 and the maximum is 4096. The default value is 100.

clErrorLevel

Recognition accuracy is defined as the percentage of times the recognizer accurately assigns a symbol to an input. There is no penalty or gain if the recognizer does not attempt a match and returns "unknown." The value can range from 0 to 100.

There are situations in which a higher accuracy rating is preferable despite an increased number of unknown results. For example, in a forms application, the Social Security field must be correctly recognized. If the recognizer is unsure, it can get the application to prompt the user again for the input (or a portion of it). At other times, it is preferable that the recognizer make a guess, no matter how wild, in order to limit the number of unknown results. For example, while taking notes in a meeting, the user may not care whether all of the results are transcribed perfectly.

The **clErrorLevel** member allows the application to signal its preference to the recognizer. Recognizers should return the SYV_UNKNOWN symbol for any symbol having a confidence level below **clErrorLevel**.

alc

The **alc** member is used to define the enabled alphabet for any **RC** structure with ALC_ constants. Any of the ALC_ constants can be combined together with a bitwise-OR operator to form the desired set of characters.

The actual characters enabled depend on the language. For example, if the user has requested French language support, the letter "è" is included in the lowercase alphabet. In the same way, "£" replaces "$" if ALC_MONETARY is set in British systems. For a list of alphabet values, see the entry for ALC_ values in Chapter 13, "Pen Application Programming Interface Constants."

Setting the RCIP_ALLANSICHAR flag in the **wIntlPreferences** member of the **RC** structure enables all characters of the appropriate set regardless of the language setting.

A recognizer that recognizes characters other than ANSI can ignore this member. If you want an application to pass character subset information to private non-ANSI recognizers, you can use the **dwRecognizer** member.

A recognizer should not return a symbol value outside the specified subset. However, a recognizer does not have to force a match to the subset; it can return SYV_UNKNOWN if a suitable match is not found.

alcPriority

The **alcPriority** member sets the priority of the ALC_ codes used to enable alphabets. It does this by telling the recognizer in which order to list options in the symbol graph.

The **alcPriority** member uses the same ALC_ codes used in the **alc** member. The bits set in **alcPriority** should be a subset of those set in **alc**. Bits set in **alcPriority** that are not also set in the **alc** member have no effect.

A recognizer can recognize a glyph that belongs to more than one enabled ALC_ subset. For example, a vertical stroke can be the letter "l" in the ALC_LCALPHA subset or the number "1" in the ALC_NUMERIC subset. The **alcPriority** member specifies that the recognizer should first return those interpretations that are in the subsets indicated in **alcPriority**. If no interpretations are in any of the **alcPriority** sets, or no priority members are set, the recognizer returns all possibilities within the enabled sets.

For example, suppose the user writes a symbol that looks like either a "q" or a "9." The generated symbol graph contains {q | 9 }. The **alcPriority** member determines the exact look of the symbol graph. If **alcPriority** has the ALC_ALPHA bit set, the recognizer should return { q | 9 } in the symbol graph. If **alcPriority** has the ALC_NUMERIC bit set, the recognizer should return { 9 | q } in the symbol graph.

Note that **alcPriority** does not affect the dictionary processing directly.

If ALC_USEBITMAP is set, the **rgbfAlc** member indicates which characters have priority.

rgbfAlc

```
#define    cbRcrgbfAlcMax          32
BYTE rgbfAlc[cbRcrgbfAlcMax];
```

The **rgbfAlc** member is the bit field used for enabled characters. For more details, see the description of ALC_ constants. If ALC_USEBITMAP is set, the 256-bit bit field in **rgbfAlc** is used to indicate which characters from the ANSI character set are currently enabled. Character 0 is the low bit of the low-order byte in the array. Characters thus indicated are connected by OR operators to any characters enabled using the other ALC_ codes. A "1" set in a bit array indicates that the character is enabled.

As an example, to enable the "$" character, set the fifth bit of byte four like this:

```
rgbfAlc[4] |= 0x10
```

A recognizer that recognizes characters other than ANSI can ignore this member. If an application wants to pass character subset information to private non-ANSI recognizers, it can use the **dwRecognizer** member of the **RC** structure.

A set of macros, defined in PENWIN.H, simplifies user setting and testing the **rgbfAlc** bits for an **RC** structure. The ANSI macros listed in the following table set (bit=1), clear (bit=0), or test (TRUE if bit==1, else FALSE) the appropriate bits in lprc->rgbfAlc corresponding to the index i, which is the ANSI value to use. The *lprc* is a pointer to the **RC** structure containing the **rgbfAlc[]** array.

Macro	Description
SetAlcBitAnsi(*lprc, i*)	Sets the bit specified by *i* in **rgbfAlc** of *lprc* to 1.
ResetAlcBitAnsi(*lprc, i*)	Resets the bit specified by *i* in **rgbfAlc** of *lprc* to 0.
IsAlcBitAnsi(*lprc, i*)	Returns TRUE if the bit specified by *i* in **rgbfAlc** of *lprc* is set.

Only the **IsAlcBitAnsi** macro returns a value (BOOL). The return values of the other macros are undefined.

Setting bits in **rgbfAlc[]** also requires combining ALC_USEBITMAP by an OR operator with **alc** for the bits to have meaning. The bits are used in addition to other **alc** settings. For example, adding ALC_NUMERIC does not also set the bits in **rgbfAlc** that correspond to 0 through 9. Thus, to recognize octal numbers (the set 0 to 7), use the following code:

```
RC rc;
int i;

rc.alc = ALC_USEBITMAP;                 // Note no ALC_NUMERIC
for (i = (int)'0'; i <= (int)'7'; i++)
     SetAlcBitAnsi( &rc, i );
```

wResultMode

The **wResultMode** member specifies the timing and granularity of the results messages to be sent back to the specified window. The following times are defined.

Constant	Description
RRM_WORD	The granularity is set at a word boundary. As soon as the recognizer sees a word break, it can send all symbols up to the point of the word break.
RRM_NEWLINE	The granularity is set at a new line. As soon as the recognizer sees a line break, it sends the result to that point.
RRM_COMPLETE	When recognition is completed by one of the methods (for example, time-out or barrel button), the results message is sent just before **Recognize** returns.
RRM_STROKE	The granularity is set at the stroke level. A result message is sent at each stroke. This is used in the NULL recognizer.
RRM_SYMBOL	The granularity is set at the symbol level. A result message is sent at each symbol. Default dictionary processing is disabled when this value is used.

A recognizer is free to send the messages any time after the requested time (defined in the preceding order), but it cannot send any messages sooner. Because of recognizer constraints, a recognizer may combine intermediate results messages. For example, if an application requests RRM_WORD, the recognizer may choose to return results on a line-by-line basis instead.

Results sent at a word boundary do not have to be sent strictly one word at a time. The requirements are as follows:

- The raw data returned must be contiguous, and it must begin with a pen-down and end with a pen-up.
- The returned "word" may contain spaces. For example, "fat{space | NULL}cat" would be resolved into two words, "fat cat." This is also necessary if the raw data for successive words overlaps.
- The recognizer should not send a word until it knows what follows the word. If the word is followed by a word on the same line, the word should be space-terminated. If the word is followed by text on a new line, the recognizer should append a soft newline symbol. The key point is that the recognizer must make it possible for the application to detect word and line spacing so it can display the recognized text appropriately.
- Once a word has been sent, the recognizer cannot change the results because of the late arrival of more strokes.

The rules for returning results with RRM_NEWLINE are similar:

- The new line should be included with the symbol graph in the result.
- Once a word has been sent, the recognizer cannot change the results because of the late arrival of more strokes.

wTimeOut

The **wTimeOut** member specifies the time-out threshold. After the time-out threshold has passed, the recognizer stops the recognition process.

Time-out occurs if more than **wTimeOut** milliseconds elapse between the most recent pen-up and the next pen-down. If time-out occurs, the recognition context is closed. Closing a recognition context means no more data is accepted; the existing data is processed and the results are sent to the application. This value is ignored if **lPcm** does not enable time-out.

In general, applications should use the value set by the user in the Control Panel. This value can be set by setting this member to RC_WDEFAULT.

The maximum value allowed is 65,534 milliseconds. If **wTimeOut** is set to 0xFFFF (65,535), the system-level value is used.

lPcmrectBound
rectExclude

These three members of the **RC** structure set the conditions for ending recognition. The **lPcm** member sets the flags for ending recognition, expressed as a bitwise-OR combination of PCM_ values.

The two **RECT** members specify inclusive and exclusive rectangles for inking. The rectangle values are in screen coordinates or, if RCO_TABLETCOORD is set, in tablet coordinates. RCO_TABLETCOORD cannot be used with **ProcessWriting**.

When **RCRESULT** is returned, the **rectBound** and **rectExclude** values are converted from screen to tablet coordinates and the RCO_TABLETCOORD flag is set.

Only pen events within **rectBound** are collected as part of the recognition context. If PCM_RECTBOUND

is set in **lPcm**, the first pen-down event outside the rectangle closes the context. Dragging the pen outside the rectangle after starting inside does not close the context; the data is still collected outside the rectangle.

If PCM_RECTEXCLUDE is set in **lPcm**, any pen-down event within **rectExclude** closes the context. The event that ends pen collection mode—that is, an event outside the bounding rectangle or inside the exclusion—is entered into Windows as a mouse event. For hit-testing the rectangles, the top and left borders are included, but not the right or bottom borders.

The bounding rectangle set by **InitRC** is valid only until the window is resized or moved. If the window is moved or resized, the application should specify again the **rectBound** member in the **RC** structure.

guide

The **guide** member is a structure of the **GUIDE** type. It contains information that specifies the placement of guidelines in the writing area for the recognizer's use.

wRcOrient

The **wRcOrient** member specifies the orientation of the tablet, expressed as RCOR_ values. For a list of orientation values, see the entry for RCOR_ values in Chapter 13, "Pen Application Programming Interface Constants."

wRcDirect

The **wRcDirect** member informs the recognizer of the direction of writing, expressed as RCOR_ values. There are both primary and secondary directions. For example, English is written from left to right (primary) and then down the page (secondary). Chinese is often written from the top down (primary) and then right to left across the page (secondary). For a list of direction values, see the entry for RCD_ values in Chapter 13, "Pen Application Programming Interface Constants."

The high byte of the direction indicates primary direction; the low byte, secondary direction. A recognizer can choose to ignore this word and support only the natural direction of the given language. The default value is determined by the recognizer.

Not all recognizers can respond to this member.

nInkWidth
rgbInk

These two members specify the ink width and color to be used during inking. The **nInkWidth** member is the thickness in pixels of the pen to use during inking. If this value is 0, no ink is drawn. The current maximum value allowed is 15. The default is the ink width set in the global **RC**.

The **rgbInk** member is the color to use for inking. If this is not a solid color, it is mapped to the closest solid color. The default is the ink color set in the global **RC**.

dwAppParam
dwRecognizer

These two members are analogous to the **dwDictParam** member described below. The **dwAppParam** value is provided for use by the application and passed to the application by way of the **lprc** member in the **RCRESULT** structure.

The **dwRecognizer** value is passed to the recognizer specified in **rc**.**hrec**. Applications can use this to pass information to a private recognizer for functionality not directly supported.

These values are set to 0 by **InitRC** and should remain 0 if they are not used by the application or recognizer.

dwDictParam

The **dwDictParam** parameter is set by an application and passed on to the dictionary by the RC Manager. It is intended to provide for dictionary functionality not directly supported. For example, a dictionary can request that the application pass in a pointer to a structure that contains a given sentence. You can use this information to extend the dictionary functionality—to highlight misspelled words, for example.

If it is not used by the application, **dwDictParam** should be left to the value (0) set by **InitRC**.

rgwReserved[cwRcReservedMax]

The **rgwReserved** member is reserved. Applications should not change the values set by **InitRC** for this member.

See Also

GUIDE, **PCMINFO**

ALC_, RCRT_, PCM_, RCD_, RCOR_

RCRESULT Overview

Overview

1.0 2.0

Applications that do not call the superseded functions **RecognizeData** or **Recognize** do not use the **RCRESULT** structure. In conforming to version 2.0 of the Pen API, applications instead use **HRCRESULT** objects, which render **RCRESULT** obsolete.

The **RCRESULT** structure is provided only for compatibility with version 1.0 of the Pen API, and will not be supported in future versions.

```
typedef struct {
     SYG syg;
     UINT wResultsType;
     int cSyv;
     LPSYV lpsyv;
     HANDLE hSyv;
     int nBaseLine;
     int nMidLine;
     HPENDATA hpendata;
     RECT rectBoundInk;
     POINT pntEnd;
     LPRC lprc;
} RCRESULT;
```

Members

syg

Symbol graph.

wResultsType

An RCRT_ value.

cSyv

Number of symbol values, not including the NULL terminator.

lpsyv

Null-terminated pointer to the recognizer's best guess.

hSyv

Globally shared handle to the symbol value specified by the **lpsyv** member.

nBaseLine

Zero or baseline of input writing.

nMidLine

Zero or midline of input writing.

hpendata

Handle to pen data.

rectBoundInk

Bounding rectangle for ink.

pntEnd

Point that terminated recognition.

lprc

Recognition context used.

Comments

When an application calls **Recognize**, **RecognizeData**, or **ProcessWriting**, the WM_RCRESULT message is sent to the appropriate window procedure when the recognizer has a result to return. The *wParam* parameter of the message contains the reason recognition ended (one of the REC_ codes). It is REC_OK if more results will be sent; otherwise, it is the same value for the last message returned by **Recognize** or **RecognizeData**. The *lParam* parameter is a far pointer to an **RCRESULT** structure. All of the data in the **RCRESULT** structure is in tablet coordinates.

The following sections elaborate on the **RCRESULT** members. All of the members are allocated with GMEM_SHARE so they can be passed between processes.

syg

This member contains the raw results returned by the recognizer. These include the various possible interpretations of the pen input, the mapping of the results to the raw data, and locations of any hot spots if the result is a gesture. The **syg**.**lpsyc** member is not valid unless RCP_MAPCHAR was set in the **RC** structure when **Recognize** or **RecognizeData** was called.

wResultsType

This member indicates the type of recognition results, expressed as a bitwise-OR combination of RCRT_ values. The RCRT_ values are not mutually exclusive. Note that the recognizer should never have to set RCRT_GESTURETOKEYS, RCRT_ALREADYPROCESSED, or RCRT_GESTURETRANSLATED. For a list of values, see the entry for RCRT_ values in Chapter 13, "Pen Application Programming Interface Constants."

lpsyv

This member contains the symbols that are recognized. An application should use these values to display the text or gestures recognized. The **lpsyv** member is the result of any dictionary search on the **SYG** structure or further postprocessing. It is NULL if the NULL recognizer is used.

hpendata

This member contains the raw data captured during inking.

rectBoundInk

This is the bounding rectangle of the ink drawn during recognition. It is in coordinates of the window that receives the results. If the user attempts to draw ink outside **rc**.**rectbound**, the ink will not be displayed. However, **rectBoundInk** is calculated as though the ink were drawn.

If data is collected outside the bounding rectangle, the **rectBound** member of **PENDATAHEADER** reflects this. (Note that **rectBound** applies only to pen-down points.) This means, however, that a portion of the **rectBoundInk** rectangle lies outside the **rc**.**rectBound** rectangle. The actual ink drawn lies in the intersection of **rectBoundInk** and the **rc**.**rectBound** rectangle. Before calculating the intersection, convert **rectBoundInk** from tablet to screen coordinates. The bounding rectangle includes the width of the ink drawn.

pntEnd

If recognition ended on a tap outside the bounding rectangle or inside the exclusive rectangle, **pntEnd** contains the coordinates of those points in display coordinates.

lprc

This is the **RC** used for recognition. Any default values (RC_WDEFAULT or RC_LDEFAULT) are replaced by the correct default value.

See Also

RC, **SYG**, RCRT_

RECTOFS Overview

Overview

1.0 2.0

Rectangle offset for nonisometric inflation of a rectangular writing area.

```
typedef struct {
     int dLeft;
     int dTop;
     int dRight;
     int dBottom;
} RECTOFS;
```

Members

dLeft

Inflation margin leftward from left side.

dTop

Inflation margin upward from top.

dRight

Inflation margin rightward from right.

dBottom

Inflation margin downward from bottom.

Comments

Inflation margins are in screen (pixel) coordinates. To inflate a window rectangle, **dLeft** and **dTop** should be negative (moving in the negative x- and y-directions, respectively) and **dRight** and **dBottom** should be positive. To deflate the rectangle, reverse the signs of the margins.

In addition to having the basic characteristics of an edit control, an hedit or bedit control must make allowances for the input of handwriting. The client rectangle often needs to be adjusted to a larger size to allow for easier writing.

For example, the cut gesture typically extends above the selected text it is deleting. If the gesture is arbitrarily clipped off at the edge of the client window, recognition accuracy suffers. Likewise, restricting handwriting input to stay within the lines can also hinder recognition accuracy. To correct this, rectangle offsets are used in hedit and bedit controls to make the writing area slightly larger than the client window size of a normal edit control. The HE_SETINFLATE and HE_GETINFLATE *wParam* values of the WM_PENCTL message are used for this purpose. These messages use a **RECTOFS** structure as a parameter. The values in the **RECTOFS** structure are added to the corresponding client area to create the bounding rectangle for the ink.

The inflation does not need to be symmetrical in every direction.

See Also

WM_PENCTL, HE_SETINFLATE, HE_GETINFLATE

SKBINFO Overview

Overview

1.0 2.0

Stores information about the current onscreen keyboard.

```
typedef struct {
      HWND hwnd;
      UINT nPad;
      BOOL fVisible;
      BOOL fMinimized;
      RECT rect;
      DWORD dwReserved;
} SKBINFO;
```

Members

hwnd

Handle of window for onscreen keyboard.

nPad

Current view of the keypad. Either SKB_FULL, SKB_BASIC, or SKB_NUMPAD for full, basic, or numeric keypad, respectively.

fVisible

TRUE if the keyboard is available and visible, FALSE otherwise.

fMinimized

TRUE if the keyboard is minimized, FALSE otherwise.

rect

Screen coordinates or the restored keyboard rectangle.

dwReserved

Must be 0.

See Also

ShowKeyboard

STRKFMT Overview

Overview

2.0

Provides a method for retrieving or changing the attributes of specified strokes in an iedit control. This structure is used by the IE_GETFORMAT and IE_SETFORMAT messages.

```
typedef struct {
     DWORD cbSize;
     UINT iesf;
     UINT iStrk;
     PENTIP tip;
     DWORD dwUser;
     DWORD dwReserved;
} STRKFMT;
```

Members

cbSize

Size of this structure in bytes.

iesf

Stroke format flags. Note that the first three flags cannot be combined with each other by the bitwise OR operator:

Constant	Description
IESF_ALL	Assume all strokes in the control.
IESF_SELECTION	Assume all selected strokes.
IESF_STROKE	Assume the stroke specified in the **iStrk** member, described in "Comments" section.
IESF_PENTIP	Set both pen tip color and width.
IESF_TIPCOLOR	Set only pen tip color.
IESF_TIPWIDTH	Set only pen tip width.

iStrk

Index of a specific stroke.

tip

PENTIP structure, containing ink tip attributes.

dwUser

User data for stroke.

dwReserved

Reserved.

Comments

When sending either IE_GETFORMAT or IE_SETFORMAT, the application initializes the **iesf** member with bit flags, indicating:

- The strokes in the iedit control to which the IE_ messages refer.
- The attributes (color and/or width) of those strokes.

Setting the IESF_STROKE bit flag in **iesf** limits action to the single stroke identified in **iStrk**. Setting IESF_SELECTION references all selected strokes. Setting IESF_ALL references all strokes in the control. These flags are mutually exclusive and cannot be combined.

The application must also set the bit flags IESF_TIPCOLOR or IESF_TIPWIDTH in the **iesf** member of the structure. These bit flags identify the stroke attribute to which the IE_ messages refer. For convenience, the defined value IESF_PENTIP combines IESF_TIPCOLOR and IESF_TIPWIDTH to identify both color and width.

With these bit flags, an application can alter stroke color and width at the same time or alter only one attribute while leaving the other unchanged.

Sending an IE_GETFORMAT message to the control produces the following results:

- If the requested strokes have the same color and width, the returned **STRKFMT** contains the common color and width in the **rgb** and **bwidth** members of its **PENTIP** structure. The return value from IE_GETFORMAT is 0.
- If the requested strokes do not all share the same attribute, the returned **STRKFMT** contains the attribute of the last stroke in the group. The return value from IE_GETFORMAT contains the bit flags IESF_TIPCOLOR and/or IESF_TIPWIDTH to indicate the attribute in which the strokes differ.

Before using **STRKFMT**, an application must initialize **cbSize** with `sizeof( STRKFMT )`.

See Also

IE_GETFORMAT, IE_SETFORMAT, **PENTIP**

STROKEINFO Overview

Overview

1.0 2.0

Contains information about a sequence of pen data.

```
typedef struct {
     UINT cPnt;
     UINT cbPnts;
     UINT wPdk;
     DWORD dwTick;
} STROKEINFO;
```

Members

cPnt

Count of points in the stroke.

cbPnts

Used internally to contain length of compressed data. Applications should ignore this value.

wPdk

State of the stroke, expressed as a PDK_ value.

dwTick

Time at beginning of the stroke. **dwTick** holds the number of milliseconds that have elapsed since the system tick reference that Windows determines at startup.

Comments

The **STROKEINFO** structure serves two main purposes. First, it is returned by the **GetPenHwEventData** functions with each piece of new data from the tablet. Second, it is used in certain pen data functions such as **AddPenInputHRC**, **AddPointsPenData**, and **GetPenDataStroke** as a header for each stroke. In both cases, it contains information about a sequence of data from the tablet.

For examples and further information about **STROKEINFO** and its members, see the section "Recognition Functions" in Chapter 8, "Writing a Recognizer."

For a list of stroke state bits, refer to the entry for PDK_ values in Chapter 13, "Pen Application Programming Interface Constants."

See Also

AddPenInputHRC, **GetPenDataStroke**, **GetPenInput**, **GetPenHwEventData**, **InsertPenDataStroke**, **TargetPoints**, PDK_

SYC Overview

Overview

1.0 2.0

The **SYC** symbol correspondence structure is best described in context with the **SYG** symbol graph and **SYE** symbol element structures. For a description of the **SYC** structure, see the entry for **SYG** below.

The **SYC** structure is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

A single shape can be identified by one or more **SYC** structures.

```
typedef struct {
     UINT    wStrokeFirst;
     UINT    wPntFirst;
     UINT    wStrokeLast;
     UINT    wPntLast;
     BOOL    fLastSyc;
} SYC;
```

Members

wStrokeFirst

Index number of the first stroke of the correspondence.

wPntFirst

Index number of the first point in the stroke identified by **wStrokeFirst**.

wStrokeLast

Index number of the last stroke of the correspondence.

wPntLast

Index number of the last point in the stroke identified by **wStrokeLast**.

fLastSyc

TRUE if there are no more **SYC** structures for the current **SYE** (symbol element).

Comments

All indexes are zero-based, so that an index of 0 indicates the first of a sequence.

Figure 11.6 illustrates the relationship of symbol values and symbol graphs. The first line shows that a symbol value is a single **SYE** symbol element. A series of symbol values can be connected by the SYV_OR value to create an OR string, as the second line illustrates. This OR string begins with the SYV_BEGINOR value and ends with a symbol value followed by SYV_ENDOR. The third line shows a symbol graph that is simply a symbol value or an OR string, in either case ending with the SYV_NULL value.

{ewc msdncd, EWGraphic, bsd23554 5 /a "SDK_5.BMP"}

See Also

RCRESULT, **TrainContext**, SYV_

SYE Overview

Overview

1.0 2.0

The **SYE** symbol element structure is best described in context with the **SYG** symbol graph and **SYC** symbol correspondence structures. For a description of the **SYE** structure, see the entry for **SYG** below.

The **SYE** structure is provided only for compatibility with version 1.0 of the Pen API, and will not be supported in future versions.

An **SYE** structure contains a symbol, which can be a character, gesture, or string.

```
typedef struct {
     SYV     syv;
     LONG    lRecogVal;
     CL      cl;
     int     iSyc;
} SYE;
```

Members

syv

Symbol value.

lRecogVal

Reserved.

cl

Confidence level.

iSyc

Index into array of **SYC** structures. The array identifies the raw data that makes up the symbol. It is possible for several **SYE** structures to use the same **SYC** structures.

SYG Overview

Overview

1.0 2.0

A symbol graph, which represents the possible interpretations identified by the recognizer.

```
typedef struct {
     POINT rgpntHotSpots[MAXHOTSPOT];
     int cHotSpot;
     int nFirstBox;
     LONG lRecogVal;
     LPSYE lpsye;
     int cSye;
     LPSYC lpsyc;
     int cSyc;
     SYV syv;
     LONG lRecogVal;
     CL cl;
     int iSyc;
     UINT wStrokeFirst;
     UINT wPntFirst;
     UINT wStrokeLast;
     UINT wPntLast;
     BOOL fLastSyc;
} SYG;
```

Members

rgpntHotSpots[MAXHOTSPOT]

Hot spots of the symbol (if any). MAXHOTSPOT is defined as 8.

cHotSpot

Number of valid hot spots in **rgpntHotSpots**.

nFirstBox

Row-major index to box of first character in result.

lRecogVal

Reserved.

lpsye

Pointer to array of **SYE** structures representing nodes of symbol graph.

cSye

Number of **SYE** structures in array **lpsye**.

lpsyc

Pointer to corresponding array of **SYC** symbol ink structures.

cSyc

Number of **SYC** structures in symbol graph.

Comments

All indexes are zero-based.

If a single entity recognized by the recognizer is mapped to a string of several symbol values, the recognizer creates multiple **SYE**. This is the case for recognizers that can recognize highly stylized sequences of characters—for example, "ing"—in which the individual characters are not necessarily recognized.

The **nFirstBox** member has no meaning for gestures. A gesture is applied to the location indicated by its hot spot.

The **SYG**, **SYE**, and **SYC** structures define the relationship between raw pen data and recognized results. However, in version 2.0 of the Pen API they are rarely of interest to applications for two reasons. First, API functions return recognition results without forcing the application to deal with the complexities of raw pen data. And second, **SYG**, **SYE**, and **SYC** apply mainly to recognizers.

All nontrivial recognizers should somehow track the pen strokes that form each character in the returned results. To be compatible with version 1.0, a recognizer must use the **SYG**, **SYE**, and **SYC** structures and return a symbol graph—an **SYG** structure—as a member of the **RCRESULT** structure. Version 2.0 does not mandate how a recognizer should map pen data to symbols. However, these three structures represent a viable method. Recognizer developers writing for version 2.0 may want to use the structures or create variations.

The following information applies to version 1.0 applications and recognizers, and to version 2.0 recognizers that employ symbol graphs to relate strokes to recognized symbols. For further information about **SYG**, **SYE**, and **SYC**, see "Returning Results" in Chapter 8, "Writing a Recognizer."

A *symbol graph* is a representation of the possible interpretations identified by the recognizer. The RC Manager processes the symbol graph using the dictionary path to identify the best interpretation. This best interpretation is returned in the results message along with the symbol graph.

A *symbol value* is a 32-bit value that represents a glyph (such as a character or a gesture) recognized by a recognizer. This is sometimes referred to as a *symbol*. A *symbol string* is an array of symbols terminated with SYV_NULL.

Each element of the symbol graph, an **SYE**, contains information about the recognized character—for example, bounding rectangle and hot spots. The **SYC** structure maps **SYE** structures back to the corresponding raw data. If two or more consecutive **SYE** structures map to the same **SYC**, they represent an indivisible unit. For example, the user might teach the system of "th" with the crossbar of the "t" connected to the "h." **SYC** structures are used primarily for training.

A version 1.0 application generally does not use the symbol graph directly. Instead, it uses the **hSyv** member of **RCRESULT**, which contains a symbol string that represents the best interpretation from the symbol graph. **SYE** and **SYC** structures work together with the **HPENDATA** memory block to identify strokes and meanings for ink. The following table lists the basic functions of these structures.

Structure	Description
HPENDATA	Contains raw data information: strokes, pen up, pen down, points, and so on.
SYC	A symbol character map. **SYC** structures delimit strokes in an **HPENDATA**. A single shape can be identified by one or more **SYC** structures. Each **SYC** identifies a starting stroke, an ending stroke, a starting point, and an ending point. A flag also indicates whether subsequent **SYC** structures in the array contain additional

	strokes for the shape. (This feature is used for delayed strokes, such as the cross stroke of the letter "t.")
SYE	A symbol element. An **SYE** contains a symbol, which can be a character, a gesture, or a string. The symbol is denoted by an **SYV**. The **SYE** contains an index into an array of **SYC** structures; this array identifies the raw data that makes up the symbol. It is possible for several **SYE**s to use the same **SYC** structures. The **SYC** structures contain indexes into the raw data.
SYV	A symbol value.
SYG	A symbol graph.

A set of **SYE**s and **SYC**s, together with an **HPENDATA** structure, is sufficient to define ink and specify how that ink should be interpreted. The training functions **TrainContext** and **TrainInk** use this information in training.

TARGET Overview

Overview

2.0

Contains information about a single target.

```
typedef struct {
      DWORD dwFlags;
      DWORD idTarget;
      HTRG htrgTarget;
      RECTL rectBound;
      DWORD dwData;
      RECTL rectBoundInk;
      RECTL rectBoundLastInk;
} TARGET;
```

Members

dwFlags

Reserved for future extensions. Must be set to 0.

idTarget

Array index to the target within **rgTarget** array in **TARGINFO** structure.

htrgTarget

Handle to the owner window that receives messages on behalf of the target.

rectBound

Bounding rectangle of the target.

dwData

Target-specific extra information to be filled during data collection.

rectBoundInk

Reserved; must be 0.

rectBoundLastInk

Reserved; must be 0.

See Also

TargetPoints, **TARGINFO**, WM_PENEVENT, **INPPARAMS**

TARGINFO Overview

Overview

2.0

A set of targets.

```
typedef struct {
     DWORD cbSize;
     DWORD dwFlags;
     HTRG htrgOwner;
     WORD cTargets;
     WORD iTargetLast;
     TARGET rgTarget[1];
} TARGINFO;
```

Members

cbSize

Size of this structure in bytes. Note that this is the original size, which assumes only a single **TARGET** structure in **rgTarget**. The value should be sizeof(TARGINFO).

dwFlags

Flags have been defined to get different targeting behavior depending on the needs of the calling application. These flags work as hints for the targeting algorithm. The flags are considered by the Pen API in the order in which they appear in the following list. If none of the flags are set, the stroke is not assigned to any target.

TPT_TEXTUAL

When this flag is set, Windows applies textual heuristics, such as identifying word breaks, while deciding the target to which a stroke should be assigned.

If there is no text to intersect with, the input is disregarded completely. Therefore, this option should not generally be used by itself.

TPT_INTERSECTINK

Indicates that if the stroke being targeted intersects with the ink in a target, the stroke should be assigned to that target. Intersection is determined based on the bounding rectangle of the stroke and the bounding rectangle of the pen data assigned to a target. If there is no ink to intersect with, the input is disregarded completely. Therefore, this option should not generally be used by itself.

TPT_CLOSEST

Indicates that the stroke should be targeted to the target closest to the stroke. The bounding rectangle of the stroke and the bounding rectangle of the target are specified by the **rectBound** element of the **TARGET** structure.

htrgOwner

Handle to the owner target. Use the **HtrgFromHwnd** and **HwndFromHtrg** macros to convert a target handle of **HTRG** type to and from an **HWND** type.

cTargets

Number of targets.

iTargetLast

Last target. Used by the **TargetPoints** function during textual heuristics. **iTargetLast** contains the

number of the target window that last received data. The system uses this value to optimize its determination of the next target. Applications can read but should not overwrite **iTargetLast**.

rgTarget[1]

Variable-length array of targets.

Comments

For best results, most applications should set all hints for targeting. That is, the **dwFlags** member of **TARGINFO** should be set to TPT_DEFAULT, which is the combination of TPT_TEXTUAL | TPT_INTERSECTINK | TPT_CLOSEST.

Before using **TARGINFO**, an application must initialize **cbSize** with sizeof(TARGINFO).

See Also

TARGET, **TargetPoints**

Pen Application Programming Interface Messages

This chapter describes in alphabetical order many of the messages and submessages defined by the Pen Application Programming Interface. Each entry describes a separate message organized under the following topic headings:

Topic heading	Description
Parameters	Message *wParam* and *lParam* parameters.
Return Value	Return value from the Windows **SendMessage** function (if applicable).
Comments	Additional information about the message.
See Also	Cross-references to related API services.

HE_CANCELCONVERT

Cancels Kana-to-Kanji conversion. Submessage of WM_PENCTL. (Japanese version only.)

Parameters

wParam

HE_CANCELCONVERT.

lParam

Reserved and must be 0.

Return Value

Returns TRUE if there are no errors; otherwise, returns FALSE.

HE_CHAROFFSET

Converts the logical character position of a character in a control into a byte offset to the character. Submessage of WM_PENCTL.

Parameters

wParam

HE_CHAROFFSET.

lParam

The low-order word contains the logical character position. The high-order word is reserved and must be 0.

Return Value

If the supplied logical character position is less than the total number of logical characters in the control, the low-order word of the return value contains the requested byte offset of the position and the high-order word is 0. Otherwise, the low-order word contains the length of the text in bytes and the high word contains 0xFFFF.

Comments

This submessage is for bedit controls only. Both the logical character position and the byte offset are zero-based.

See Also

WM_PENCTL

HE_CHARPOSITION

Converts a byte offset in the text buffer of a control to the logical character position, which contains the byte specified by the byte offset. Submessage of WM_PENCTL.

Parameters

wParam

HE_CHARPOSITION.

lParam

The low-order word contains the byte offset. The high word is reserved and must be 0.

Return Value

If the supplied byte offset is less than the length of the text in bytes, the low-order word of the return value contains the logical character position and the high-order word is 0; otherwise, the low-order word contains the total number of logical characters in the text of the control and the high-order word contains 0xFFFF.

Comments

This submessage is for bedit controls only. Both the byte offset and the logical character position are zero-based.

See Also

HE_CHAROFFSET, WM_PENCTL

HE_DEFAULTFONT

Switches the font of a bedit control to the default font that the bedit control selected at the time of creation. Submessage of WM_PENCTL.

Parameters

wParam

HE_DEFAULTFONT.

lParam

If the low-order word is nonzero, the control is repainted.

Return Value

The return value is undefined.

Comments

This submessage is for bedit controls only.

See Also

WM_PENCTL

HE_ENABLEALTLIST

Enables or disables the alternate list in a bedit control. Submessage of WM_PENCTL.

Parameters

wParam

HE_ENABLEALTLIST.

lParam

If the low-order word is nonzero, the alternate list menu is enabled; otherwise, it is disabled.

Return Value

The return value is undefined.

Comments

This submessage is for bedit controls only.

See Also

HE_HIDEALTLIST, HE_SHOWALTLIST, WM_PENCTL

HE_FIXKKCONVERT

Confirm undetermined string and close Input Method Editor (IME). Submessage of WM_PENCTL. (Japanese version only.)

Parameters

wParam

HE_FIXKKCONVERT.

lParam

Reserved and must be 0.

Return Value

Returns TRUE if there are no errors; otherwise, returns FALSE.

Comments

When in preconversion mode, the marked conversion string area is removed. Marked conversion strings are indicated in the Input Method Editor (IME) in a different shade than standard edit control text selection. When in conversion mode (no marked string), the string is confirmed and the IME is closed. Available for bedits only.

HE_GETBOXLAYOUT

Retrieves the box layout for a control. Submessage of WM_PENCTL.

Parameters

wParam

HE_GETBOXLAYOUT.

lParam

Address of a **BOXLAYOUT** structure that is filled with the current box layout for the control.

Return Value

The return value is undefined.

Comments

This submessage is for bedit controls only.

See Also

BOXLAYOUT, WM_PENCTL

HE_GETCONVERTRANGE

Gets the range of the marked conversion string. Submessage of WM_PENCTL. (Japanese version only.)

Parameters

wParam

HE_GETCONVERTCHAR.

lParam

Not used.

Return Value

Returns a 32-bit value with the starting character position (not byte position) of the marked conversion string in the low-order word and the position of the last character of the marked conversion string plus 1 in the high-order word.

Comments

Available for bedits only. The message returns a valid value only when in preconversion mode; otherwise, it returns 0.

HE_GETINFLATE

Retrieves the inflation rectangle for a control. Submessage of WM_PENCTL.

Parameters

wParam

HE_GETINFLATE.

lParam

Address of a **RECTOFS** structure that is filled with the current values for the inflation rectangle.

Return Value

Returns TRUE if successful; otherwise, FALSE.

Comments

For a description of how to increase or decrease the writing area of a control, see "HE_SETINFLATE Submessage" in Chapter 3, "The Writing Process."

See Also

HE_SETINFLATE, **RECTOFS**, WM_PENCTL

HE_GETINKHANDLE

Retrieves the ink handle for the current control. Submessage of WM_PENCTL.

Parameters

wParam

HE_GETINKHANDLE.

lParam

Unused.

Return Value

The low-order word of the return value contains a handle to the captured ink. If the return value is NULL, the control is not in ink mode.

Comments

The returned ink handle is valid only during the life of the control. The handle becomes invalid after the control is destroyed.

See Also

WM_PENCTL

HE_GETKKCONVERT

Determines if the Input Method Editor (IME) is in conversion mode. Submessage of WM_PENCTL. (Japanese version only.)

Parameters

wParam

HE_GETKKCONVERT.

lParam

Reserved and must be 0.

Return Value

Returns TRUE if the IME is in conversion mode; otherwise, returns FALSE.

Comments

Available for bedits only.

HE_GETKKSTATUS

Determines the mode of the Kana-to-Kanji conversion in the Input Method Editor (IME). Submessage of WM_PENCTL. (Japanese version only.)

Parameters

wParam

HE_GETKKSTATUS.

lParam

Reserved and must be 0.

Return Value

Returns one of the following values or FALSE:

Value	Current Status
HEKKR_PRECONVERT	In preconversion mode.
HEKKR_CONVERT	In conversion mode
HHEKKR_NOCONVERT	In nonconversion mode

Comments

The IME recognizes three modes: preconversion mode, conversion mode, and non-conversion mode. In preconversion mode, the user has entered text intended for conversion by the IME and the text is marked. When the user invokes the IME on the marked range of characters, the conversion mode is entered and the IME is active. Once the user confirms an IME conversion, the nonconversion mode (normal mode) is entered.

The term "marked" refers to the range of cells in the bedit that have been selected for character conversion. Characters marked for conversion appear differently to the user than normally selected characters.

This submessage is available for bedits only. This message can also be used to determine keyboard IME status by checking for HEKKR_CONVERT.

HE_GETUNDERLINE

Queries for the current underline mode. Submessage of WM_PENCTL.

Parameters

wParam

HE_GETUNDERLINE.

lParam

Unused.

Return Value

Returns TRUE if the underline mode is set; otherwise, FALSE.

Comments

This submessage is for hedit controls only.

See Also

WM_PENCTL

HE_HIDEALTLIST

Hides the alternate list in a bedit control, assuming an alternate list menu is being displayed. Submessage of WM_PENCTL.

Parameters

wParam

HE_HIDEALTLIST.

lParam

If the low-order word is HEAL_DEFAULT, the alternate list menu is hidden.

Return Value

The return value is undefined.

Comments

This submessage is for bedit controls only.

See Also

HE_ENABLEALTLIST, HE_SHOWTLIST, WM_PENCTL

HE_KKCONVERT

Starts Kana-to-Kanji conversion. Submessage of WM_PENCTL. (Japanese version only.)

Parameters

wParam

HE_KKCONVERT.

lParam

Must be one of the following values:

Value	Meaning
HEKK_DEFAULT	The first time the conversion is specified, the selected character string is replaced with the conversion result; the second time it is specified, the conversion candidate dialog box appears.
HEKK_CONVERT	The selected character string is replaced with the conversion result regardless of how many times conversion has been specified.
HEKK_CANDIDATE	Causes the conversion candidate dialog box to appear.
HEKK_DBCSCHAR	The SBCS characters (0x20 - 0x7E, 0xA1 - 0xDF) are replaced by their DBCS equivalents.
HEKK_SBCSCHAR	The DBCS characters in the selected character string or marked conversion string that have equivalent SBCS representations are replaced by their equivalent SBCS characters.
HEKK_HIRAGANA	The katakana characters (DBCS or SBCS) in the selected character string or marked conversion string are replaced with their hiragana equivalents.
HEKK_KATAKANA	The hiragana characters in the selected character string or marked conversion string are replaced with their DBCS katakana representation.

Return Value

Returns TRUE if there are no errors; otherwise, returns FALSE:

Comments

In this message, "marked conversion string" indicates the string in the Input Method Editor (IME) that is marked for conversion. Text marked for conversion is indicated by a different selection color than that used for normal text selection in a standard text edit control. Available for bedits only.

HE_PUTCONVERTCHAR

Sends a character to the Input Method Editor (IME) and marks it for conversion. Submessage of WM_PENCTL. (Japanese version only.)

Parameters

wParam

HE_PUTCONVERTCHAR.

lParam

The low-order word contains the character code, which can be an SBCS or DBCS character.

Return Value

Returns TRUE if there are no errors; otherwise, returns FALSE.

Comments

Posting this message is exactly like posting a WM_CHAR message to a bedit or edit control with the exception that the posted character also acquires the attribute of being a character marked for conversion in the Input Method Editor. This sub-message is available for bedits only.

HE_SETBOXLAYOUT

Sets the box layout for a control. Submessage of WM_PENCTL. Submessage of WM_PENCTL.

Parameters

wParam

HE_SETBOXLAYOUT.

lParam

Address of the **BOXLAYOUT** structure to be set.

Return Value

Returns TRUE if successful; otherwise, returns FALSE.

Comments

This submessage is for bedit controls only.

See Also

WM_PENCTL

HE_SETCONVERTRANGE

Sets the range of the marked conversion string. Submessage of WM_PENCTL. (Japanese version only.)

Parameters

wParam

HE_SETIMEDEFAULT.

lParam

The low-order word contains the starting character position (not byte position) of the marked conversion string. The high-order word contains the ending character position plus 1.

Return Value

Returns TRUE if there are no errors; otherwise, returns FALSE.

Comments

Available for bedits only. If the starting character position is 0 and the ending char-acter position is -1, all the text in the control becomes the marked conversion string. If the starting character is -1, the marked conversion string area is removed. When characters are marked for conversion, the Input Method Editor is said to be in preconversion mode.

Returns FALSE if in conversion mode. If there is a selection, the selection will be cleared. The caret will be moved to the end of the marked conversion string.

HE_SETINFLATE

Sets the inflation rectangle for a control. Submessage of WM_PENCTL.

Parameters

wParam

HE_SETINFLATE.

lParam

Address of a **RECTOFS** structure specifying the inflation margins for the writing window.

Return Value

Returns TRUE if successful, or FALSE if an invalid window rectangle is specified.

Comments

This is a submessage of the WM_PENCTL message.

For a description of how to increase or decrease the writing area of a control, see "HE_SETINFLATE Submessage" in Chapter 3, "The Writing Process."

See Also

HE_GETINFLATE, **RECTOFS**, WM_PENCTL

HE_SETINKMODE

Starts ink data collection in a control. Submessage of WM_PENCTL.

Parameters

wParam

HE_SETINKMODE.

lParam

The low-order word is the initial **HPENDATA** object or NULL. If the initial **HPENDATA** is supplied, it must be relative to the top-left corner of the client rectangle of the control.

Return Value

Returns TRUE if successful; otherwise, returns FALSE.

Comments

This is a submessage of the WM_PENCTL message.

See Also

WM_PENCTL

HE_SETUNDERLINE

Sets or cancels underline mode in an hedit control. Submessage of WM_PENCTL.

Parameters

wParam

HE_SETUNDERLINE.

lParam

The low-order word is TRUE to set underline mode and FALSE to cancel it.

Return Value

Returns the current underline mode.

Comments

This submessage is for single-line hedit controls only.

See Also

WM_PENCTL

HE_SHOWALTLIST

Displays the alternate list menu for the current cell in a bedit control, assuming that alternate lists are enabled. Submessage of WM_PENCTL.

Parameters

wParam

HE_SHOWALTLIST.

lParam

If the low-order word is HEAL_DEFAULT, the alternate list menu is displayed.

Return Value

If more than one character is selected, the alternate list of the first character in the selection is displayed. If nothing is selected, the alternate list for the character to the right of the caret is displayed and the return value is TRUE.

Comments

This submessage is for bedit controls only. If more than one box is selected, the HE_SHOWALTLIST message will drop a word alternate list menu; otherwise, it will drop a character alternate list menu.

See Also

HE_ENABLEALTLIST, HE_HIDEALTLIST, WM_PENCTL

HE_STOPINKMODE

Stops ink collection in a control. Submessage of WM_PENCTL.

Parameters

wParam

HE_STOPINKMODE.

lParam

If the low-order word is HEP_RECOG, the control performs recognition and displays text. If the low-order word is HEP_NORECOG (0), the control removes the ink without performing recognition. If the low-order word is HEP_WAITFORTAP, the control performs recognition when the next tap in the writing area occurs.

Return Value

Returns TRUE if successful; otherwise, returns FALSE.

Comments

This is a submessage of the WM_PENCTL message.

See Also

WM_PENCTL

HN_BEGINDIALOG

Sent by a bedit or hedit control to its parent window just before the control puts up any kind of dialog, including the lens, edit text, or garbage-detection dialogs.

The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam

Specifies the identifier of the hedit or bedit control.

lParam

Specifies the handle of the hedit or bedit control in the low-order word and the HN_BEGINDIALOG notification message in the high-order word.

Return Value

If the parent window returns TRUE to this notification message, the bedit or hedit control refrains from opening the dialog; otherwise, the dialog is opened. Note that the application can disable the hedit or bedit control's ability to open a dialog by specificying CIH_NOEDITTEXT in the WM_CTLINIT message.

HN_ENDDIALOG

Sent by a bedit or hedit control to its parent window of the dialog when a dialog opened by the control is destroyed.

The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam

Specifies the identifier of the hedit or bedit control.

lParam

Specifies the handle of the hedit or bedit control in the low-order word and the HN_ENDDIALOG notification message in the high-order word.

Return Value

The return value is ignored.

HN_ENDKKCONVERT

Sent after a bedit control has completed the Kana-to-Kanji conversion. Submessage of WM_PENCTL. (Japanese version only.)

The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam

Specifies the identifier of the hedit or bedit control.

lParam

Specifies the handle of the hedit or bedit control in the low-order word and the HN_ENDKKCONVERT notification message in the high-order word.

Return Value

The return value is ignored.

HN_ENDREC

Sent after an hedit or bedit control has acted upon the results of recognition from a recognition session.

The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam

Specifies the identifier of the hedit or bedit control.

lParam

Specifies the handle of the hedit or bedit control in the low-order word and the HN_ENDREC notification message in the high-order word.

Return Value

The return value is ignored.

HN_RESULT

Sent when an hedit or bedit control receives results of inking or recognition from a recognition session, but before the control absorbs the results into its internal data structures.

The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam

Specifies the identifier of the hedit or bedit control.

lParam

Specifies the handle of the hedit or bedit control in the low-order word and the HN_RESULT notification message in the high-order word.

Return Value

The return value is ignored.

Comments

The application can get the results and optionally modify them using the WM_PENMISC message with the PMSC_GETSYMBOLCOUNT, PMSC_GETSYMBOLS, and the PMSC_SETSYMBOLS submessages.

See Also

WM_PENMISC

IE_CANUNDO

Retrieves an indication of whether the control can undo the last user operation.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns one of the following values:

Constant	Description
IER_YES	The control can undo the last operation.
IER_NO	The control cannot undo the last operation or the control has security protection disallowing an undo operation.
IER_PARAMERR	*wParam* or *lParam* is invalid.

IE_DOCOMMAND

Causes an ink edit control to execute a command.

Parameters

wParam

Contains one of the following command message codes:

Constant	Description
IEM_CLEAR	Clear (delete) the selection.
IEM_COPY	Copy selected strokes.
IEM_CUT	Cut selected strokes.
IEM_ERASE	Use eraser mode to erase.
IEM_LASSO	Use lasso mode to select strokes.
IEM_PASTE	Paste Clipboard contents to the iedit control.
IEM_PROPERTIES	Invoke the properties dialog box on the selected strokes.
IEM_RESIZE	Resize selected strokes.
IEM_SELECTALL	Select all the strokes in the control.
IEM_UNDO	Undo the last action.

lParam

Not used; must be 0.

Return Value

Returns IER_OK if successful; otherwise, returns one of the following:

Constant	Description
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_MEMERR	A memory error occurred.
IER_SECURITY	The control has security protection disallowing the operation.

Comments

An application can use this message to force the ink edit control to execute a valid command. For example, an application might have a toolbar button or menu item that can be used to trigger a copy command. The IE_DOCOMMAND message can be used in response to the user's pressing the button or selecting the menu item to have the iedit control copy the selected ink to the Clipboard.

The iedit control sends its parent an IN_COMMAND notification if the IEN_EDIT notify bit is set, to which the parent can respond in the usual ways. Sending a command that the iedit control cannot interpret (that is, a command code of IEM_USER or above) causes any specified notification but the iedit control takes no other action.

The IEN_EDIT bit is set by default. It should be cleared if the control's parent does not want to receive the IN_COMMAND notification message.

See Also

IE_GETCOMMAND

IE_EMPTYUNDOBUFFER

Empties the undo buffer.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns one of the following:

Constant	Description
IER_OK	Success.
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_SECURITY	The control has security protection disallowing the operation.

Comments

If there is nothing in the undo buffer, this message returns IER_OK but does nothing else. As long as the buffer remains empty after sending IE_EMPTYUNDOBUFFER, the messages IE_CANUNDO and WM_UNDO return FALSE.

See Also

IE_CANUNDO

IE_GETAPPDATA

Retrieves the application data saved in the ink edit control.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns the contents of the application data area if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

An application can save any DWORD value with the ink edit control. The control does not use this data. The IE_SETAPPDATA and IE_GETAPPDATA messages provide the only means for an application to interact with the data.

See Also

IE_SETAPPDATA

IE_GETBKGND

Retrieves the current background painting options of an ink edit control.

Parameters

wParam

Not used; must be 0.

lParam

Address of a WORD variable that receives the current background options, as given in the following list:

Constant	Description
IEB_BIT_CENTER	Center supplied bitmap in control.
IEB_BIT_STRETCH	Stretch bitmap to fit control.
IEB_BIT_TILE	Tile supplied bitmap repeatedly in control.
IEB_BIT_UL	Align supplied bitmap to upper-left corner in the control. (UL stands for "upper left.")
IEB_BRUSH	Use application-supplied brush in *lParam*.
IEB_DEFAULT	Do default background (use COLOR_WINDOW).
IEB_OWNERDRAW	Parent will draw background.

Return Value

If successful, returns a handle to the background bitmap or a brush, or NULL, according to the option specified in *lParam*; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The returned handle is owned by the iedit control; the application should not delete it. If the application needs to preserve this information, it should copy the handle.

See Also

IE_SETBKGND, IE_GETGRIDPEN, IE_SETGRIDPEN

IE_GETCOMMAND

Retrieves the menu item number of a selected command.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns the menu item number if successful; otherwise, returns one of the following:

Constant	Description
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_NOCOMMAND	Attempt to issue IE_GETCOMMAND when no command was selected.

Comments

The application sends the IE_GETCOMMAND message when it receives an IN_COMMAND notification to find out what menu item the user selected. This message can be sent only during processing of an IN_COMMAND notification. It returns the IER_NOCOMMAND error code if sent at any other time.

See Also

IE_DOCOMMAND

IE_GETCOUNT

Retrieves the count of strokes in the control.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

If successful, returns the total number of strokes in the control; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

See Also

IE_GETSELCOUNT

IE_GETDRAWOPTS

Retrieves the ink-drawing option.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns one of the following ink-drawing options:

Constant	Description
IEDO_FAST	Drawing is done as fast as possible. This is the default setting.
IEDO_NONE	No drawing is done (disabled drawing).
IEDO_SAVEUPSTROKES	Save pen-up strokes in the **HPENDATA** object.

Comments

IEDO_FAST and IEDO_NONE are mutually exclusive options.

See Also

IE_SETDRAWOPTS

IE_GETERASERTIP

Retrieves the eraser pen tip.

Parameters

wParam

Not used; must be 0.

lParam

Address of a **PENTIP** structure.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The **PENTIP** structure is filled with the current pen tip used for erasing.

See Also

PENTIP, IE_SETERASERTIP

IE_GETFORMAT

Retrieves the current format of a stroke or a set of strokes in an iedit control.

Parameters

wParam

Not used; must be 0.

lParam

Address of a **STRKFMT** structure.

Return Value

If successful, returns bit flags that indicate whether the strokes identified in the **STRKFMT** structure have different color or width, as described in the "Comments" section.

If an error occurs, returns one of the following values:

Constant	Description
IER_ERROR	Unknown error.
IER_PARAMERR	*wParam* or *lParam* is invalid. Also returned when there is an invalid stroke index and the IESF_STROKE option is specified in the **iesf** member of the **STRKFMT** structure.
IER_SECURITY	The control has security protection disallowing the operation.
IER_SELECTION	No valid selection when the IESF_SELECTION option is specified in the **iesf** member of the **STRKFMT** structure.

Comments

If the value in the **iesf** member of the **STRKFMT** structure has the IESF_STROKE bit set, IE_GETFORMAT refers to the single stroke identified in the **iStrk** member. In this case:

- The return value is 0.
- The **bwidth** and **rgb** members in **PENTIP** specified in the **STRKFMT** structure contain the stroke's color and width.

If either the bit IESF_SELECTION or IESF_ALL is set in **iesf**, IE_GETFORMAT retrieves format information for multiple strokes. In this case, the return value con-tains the IESF_TIPCOLOR or IESF_TIPWIDTH bit flags that indicate whether the multiple strokes share the same ink color and width.

For example, if the multiple requested strokes all have the same width, then

- The IESF_TIPWIDTH bit of the return value is 0 to indicate the strokes all have the same width.
- The **bwidth** member in **PENTIP** specified in the **STRKFMT** structure contains the common width.

If the strokes do not all have the same color, IE_GETFORMAT returns the following information:

- The IESF_TIPCOLOR bit is set in the return value to indicate the strokes do not share a common color.

- The **rgb** member in **PENTIP** specified in the **STRKFMT** structure contains the color of the last stroke in the group.

The caller must initialize the **cbSize** member of the **STRKFMT** structure to sizeof(STRKFMT) before sending IE_GETFORMAT.

The supplied **STRKFMT** structure specifies the stroke or strokes for which the attributes are desired. The structure is filled according to the request and the actual stroke attributes.

See Also

IE_SETFORMAT, **PENTIP**, **STRKFMT**

IE_GETGESTURE

Retrieves the specifics of a gesture.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns an **HRCRESULT** of the gesture if successful; otherwise, returns one of the following:

Constant	Description
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_NOGESTURE	Indicates an attempt to issue IE_GETGESTURE when no gesture was performed.

Comments

An application sends IE_GETGESTURE when it receives an IN_GESTURE notifi-cation, to retrieve the specifics of the user's gesture. This message can be sent only during processing of an IN_GESTURE notification. It returns the error code IER_NOGESTURE if it is set at any other time.

If successful, the application receives an **HRCRESULT**, which can then be used to get information about the gesture specifics. This handle is still owned by the iedit control, however, and the application must neither delete the handle nor modify the data to which it refers.

IE_GETGRIDORIGIN

Retrieves the current origin of the rule or grid-line settings for the control.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns the x-coordinate of the origin in the low-order word and the y-coordinate of the origin in the high-order word, if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

See Also

IE_GETBKGND, IE_GETGRIDORIGIN, IE_GETGRIDSIZE, IE_SETBKGND, IE_SETGRIDORIGIN, IE_SETGRIDPEN, IE_SETGRIDSIZE

IE_GETGRIDPEN

Retrieves the current GDI pen used to draw the rules or grid lines for the control.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

If successful, returns a handle to a GDI pen that is being used to draw the grid lines. This handle can be NULL; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The handle of the GDI pen returned remains the property of the iedit control. The application must not delete this handle.

See Also

IE_GETBKGND, IE_GETGRIDORIGIN, IE_GETGRIDSIZE, IE_SETBKGND, IE_SETGRIDORIGIN, IE_SETGRIDPEN, IE_SETGRIDSIZE

IE_GETGRIDSIZE

Retrieves the current horizontal and vertical spacing of the rule or grid-line settings for the control.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

If successful, the low-order word has the horizontal spacing and the high-order word has the vertical spacing; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

See Also

IE_GETBKGND, IE_GETGRIDORIGIN, IE_GETGRIDPEN, IE_SETBKGND, IE_SETGRIDORIGIN, IE_SETGRIDPEN, IE_SETGRIDSIZE

IE_GETINK

Retrieves the contents of an ink edit control.

Parameters

wParam

IEGI_ALL to get the entire ink, or IEGI_SELECTION to get only the selected ink.

lParam

Not used; must be 0.

Return Value

Returns the handle to the **HPENDATA** structure if successful; otherwise, returns one of the following:

Constant	Description
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_MEMERR	A memory error occurred.
IER_SECURITY	The control has security protection disallowing the operation.
IER_SELECTION	Nothing is selected in the control; operation assumes a selection.

Comments

The returned **HPENDATA** structure becomes the property of the application, which must eventually destroy it. This handle is a copy of the handle used internally by the control. An application cannot change the control by modifying the pen data referred to by this handle, although the modified handle can subsequently be used in an IE_SETINK call, which modifies the control's contents.

See Also

IE_SETINK

IE_GETINKINPUT

Retrieves the current ink input options for the control.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns the current ink input bits if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The ink input bits are as follows:

Constant	Description
IEI_MOVE	Move all ink inside the control.
IEI_RESIZE	Resize all ink to fit inside the control.
IEI_CROP	Crop all ink that falls outside the control.
IEI_DISCARD	Discard all ink if any falls outside the control.

See Also

IE_SETINKINPUT

IE_GETINKRECT

Retrieves the bounding rectangle of the ink.

Parameters

wParam

Not used; must be 0.

lParam

Address of a **RECT** structure.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The **RECT** structure is filled with the bounding rectangle of the current ink in the control. The rectangle is in the same coordinates as the scaling mode the **HPENDATA** object is in.

IE_GETMENU

Retrieves a handle to an ink edit control's pop-up menu.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns the handle to the menu if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The iedit control continues to own the menu handle.

The application can perform standard menu operations upon the returned handle, including the addition, deletion, and modification of menu items. The application's changes are reflected the next time the pop-up menu is invoked.

IE_GETMODE

Retrieves the current mode the control is in.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

If successful, returns one of the following values indicating the current control mode:

Constant	Description
IEMODE_READY	The control is ready for inking, moving strokes, tapping, resizing, and so on.
IEMODE_ERASE	The control is set to erasing mode.
IEMODE_LASSO	The control is set to lasso selection mode.

Otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

See Also

IE_SETMODE

IE_GETMODIFY

Queries whether the contents of the control have been modified since the control was created.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns one of the following values:

Constant	Description
IER_YES	The control's contents have been modified.
IER_NO	The control's contents have not been modified.
IER_PARAMERR	*wParam* or *lParam* is invalid.

Comments

This command succeeds regardless of the security setting.

See Also

IE_SETMODIFY

IE_GETNOTIFY

Retrieves the current notification options for the control.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

If successful, returns the current notification bits, as listed here:

Constant	Description
IEN_EDIT	Require notifications of editing or command events.
IEN_FOCUS	Require notifications of focus events.
IEN_PAINT	Require notifications of painting events.
IEN_PDEVENT	Require notifications of pointing-device events (clicks and taps).
IEN_PROPERTIES	Require notifications before bringing up the properties dialog box.
IEN_SCROLL	Require notifications of scrolling events.

Otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

See Also

IE_SETNOTIFY

IE_GETPAINTDC

Retrieves the handle to the device context (**HDC**), which is used to paint an ink edit control. This **HDC** was supplied to the iedit control by **BeginPaint**; therefore, its clipping region is set according to those portions of the iedit control that have been invalidated.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

Returns the **HDC** if successful; otherwise, returns one of the following:

Constant	Description
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_NOTINPAINT	Attempted IE_GETPAINTDC outside of paint notification.

Comments

The application can send this message only when the parent window is processing one of the painting notifications: IN_PREPAINT, IN_PAINT, IN_POSTPAINT, or IN_ERASEBKGND. An attempt to send it at any other time will fail, returning IER_NOTINPAINT.

The clipping region is already appropriately set when sending IE_GETPAINTDC. The **HDC** is in the MM_TEXT mapping mode. The **HDC** must not be released; the iedit control does this after returning from the painting notification.

IE_GETPDEVENT

Retrieves the pointing-device event that triggered the IN_PDEVENT notification. This can be from a mouse, pen, or other device.

Parameters

wParam

Not used; must be 0.

lParam

Address of a **PDEVENT** structure that is filled by the control when it receives this message.

Return Value

Returns one of the following:

Constant	Description
IER_OK	Success.
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_NOPDEVENT	No event occurred.

Comments

This message can succeed only during the processing of an IN_PDEVENT notification. At all other times its use is invalid. The caller must initialize the **cbSize** member of the **PDEVENT** structure to sizeof(PDEVENT) before sending this message.

The application can cause the event to be discarded by returning TRUE to the IN_PDEVENT notification.

See Also

IN_PDEVENT, **PDEVENT**

IE_GETPENTIP

Retrieves the default ink pen tip.

Parameters

wParam

Not used; must be 0.

lParam

Address of a **PENTIP** structure.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The **PENTIP** structure is filled with the current pen tip used for default inking.

See Also

PENTIP, IE_SETPENTIP

IE_GETRECOG

Retrieves the current recognition setting of the control.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

If successful, returns the following recognition flags, which can be combined using the bitwise-OR operator:

Constant	Description
IEREC_ALL	All recognition enabled.
IEREC_GESTURE	Gesture recognition enabled.
IEREC_NONE	Recognition disabled.

Otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

Currently, IEREC_GESTURE and IEREC_ALL are equivalent.

See Also

IE_SETRECOG

IE_GETSECURITY

Retrieves the current security setting of the control.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

If successful, returns the following security flags, which can be combined using the bitwise-OR operator:

Constant	Description
IESEC_NOCOPY	Copying not allowed.
IESEC_NOCUT	Cutting, deleting, and clearing not allowed.
IESEC_NOPASTE	Pasting disabled.
IESEC_NOUNDO	Undo not permitted.
IESEC_NOINK	Inking not allowed.
IESEC_NOERASE	Erasing not allowed.
IESEC_NOGET	The IE_GETINK message is disabled.
IESEC_NOSET	The IE_SETINK message is disabled.

Otherwise, returns IER_SECURITY to indicate that the control has security pro-tection disallowing the operation.

See Also

IE_SETSECURITY

IE_GETSEL

Retrieves the selection status of a particular stroke.

Parameters

wParam

Contains the zero-based index of the stroke whose selection status is queried.

lParam

Not used; must be 0.

Return Value

Returns one of the following values:

Constant	Description
IER_YES	The stroke is selected.
IER_NO	The stroke is not selected.
IER_PARAMERR	*wParam* or *lParam* is invalid.

See Also

IE_SETSEL

IE_GETSELCOUNT

Retrieves the number of selected strokes.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

If successful, returns the number of selected strokes; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

See Also

IE_GETSEL, IE_GETCOUNT

IE_GETSELITEMS

Retrieves a list of all selected strokes in the control.

Parameters

wParam

Size of the buffer passed in.

lParam

Address of a buffer of UINT variables that will be filled with the indices of the selected strokes in the control. This buffer must be large enough to hold all requested indices. The application can ensure this by first getting the number of selected strokes with the IE_GETSELCOUNT message and then calculating the required size of the buffer.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

See Also

IE_GETSELCOUNT

IE_GETSTYLE

Retrieves the current style attributes of an ink edit control.

Parameters

wParam

Not used; must be 0.

lParam

Not used; must be 0.

Return Value

If successful, returns the following current style bits:

Constant	Description
IES_BORDER	Border drawn around control.
IES_HSCROLL	Horizontally scrollable control.
IES_VSCROLL	Vertically scrollable control.
IES_OWNERDRAW	Application will do all ink drawing.

Otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

IE_SETAPPDATA

Sets the application data saved in the ink edit control.

Parameters

wParam

Not used; must be 0.

lParam

Data to be saved.

Return Value

Returns the previous contents of the application data area, if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

An application can save any DWORD value with the ink edit control. The control does not use this data. The IE_SETAPPDATA and IE_GETAPPDATA messages provide the only means for an application to interact with the data.

See Also

IE_GETAPPDATA

IE_SETBKGND

Sets the background painting options for the control.

Parameters

wParam

Specifies the background options, as given in the following list:

Constant	Description
IEB_BIT_CENTER	Center bitmap in control. The *lParam* parameter contains bitmap.
IEB_BIT_STRETCH	Stretch bitmap to fit control. The *lParam* parameter contains bitmap.
IEB_BIT_TILE	Tile supplied bitmap repeatedly in control. The *lParam* parameter contains bitmap.
IEB_BIT_UL	Align supplied bitmap to upper-left corner in the control. (UL stands for "upper left.") The *lParam* parameter contains bitmap.
IEB_BRUSH	Use brush supplied in *lParam*.
IEB_DEFAULT	Do default background (use COLOR_WINDOW).
IEB_OWNERDRAW	Parent will draw background.

lParam

Contains a handle to the background bitmap or a brush, or is NULL, according to the specified option in *wParam*.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The application can change the background at any time. The control is synchro-nously repainted upon any change.

The **HBITMAP** or **HBRUSH** handle, if there is one, becomes the property of the iedit control. The application must make no further use of the handle if the message returns IER_OK. If the IEB_OWNERDRAW option is selected, the parent window must process the IN_ERASEBKGND notification. If an application must place such objects as icons or metafiles in the background, it must do so either in an owner-draw capacity or during the IN_PREPAINT notification.

See Also

IE_GETBKGND, IE_GETGRIDPEN, IE_SETGRIDPEN

IE_SETDRAWOPTS

Sets the ink drawing option.

Parameters

wParam

Contains the drawing option.

Constant	Description
IEDO_FAST	Drawing is done as fast as possible. This is the default setting.
IEDO_NONE	No drawing is done (disabled drawing).
IEDO_SAVEUPSTROKES	Save pen-up strokes in the **HPENDATA** object.

lParam

Not used; must be 0.

Return Value

Returns IER_OK if successful; the previous draw option is returned in the high-order word; otherwise, returns one of the following:

Constant	Description
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_OWNERDRAW	The control is an owner-draw control; setting draw options is invalid.
IER_SECURITY	The control has security protection disallowing the operation.

Comments

Unlike using the WM_SETREDRAW message, IEDO_NONE controls only the drawing of the ink. The control's background, grid lines, and so forth are redrawn as usual. By default, pen-up strokes are not saved in the **HPENDATA** object.

IE_SETERASERTIP

Sets the eraser pen tip.

Parameters

wParam

Not used; must be 0.

lParam

Address of a **PENTIP** structure.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The pen tip specified by **PENTIP** is used for erasing in the control.

See Also

PENTIP, IE_GETERASERTIP

IE_SETFORMAT

Sets the format of a stroke or a set of strokes in an ink edit control.

Parameters

wParam

Not used; must be 0.

lParam

Address of a **STRKFMT** structure.

Return Value

Returns IER_OK if successful; otherwise, returns one of the following:

Constant	Description
IER_ERROR	Unknown error.
IER_PARAMERR	*wParam* or *lParam* is invalid. Also returned when there is an invalid stroke index and the IESF_STROKE option is specified in the **iesf** member of the **STRKFMT** structure.
IER_MEMERR	A memory error occurred.
IER_SECURITY	The control has security protection disallowing the operation.
IER_SELECTION	No valid selection with the IESF_SELECTION option in the **iesf** member of the **STRKFMT** structure.

Comments

The stroke or strokes indicated by the **STRKFMT** structure are modified as indicated and repainted (unless drawing has been turned off using the IEDO_NONE bit in IE_SETDRAWOPTS).

The **iesf** member of the **STRKFMT** structure contains the IESF_TIPCOLOR or IESF_TIPWIDTH bit flags to selectively adjust the color or width attributes of the ink. This allows setting only the color, for example, while leaving the width unchanged. If the value in **iesf** has either IESF_TIPCOLOR or IESF_TIPWIDTH set, the ink in the control adopts the new color or width given in the **rgb** or **bwidth** members of the **PENTIP** structure identified in the **tip** member of **STRKFMT**.

The caller must initialize the **cbSize** member of the **STRKFMT** structure to sizeof(STRKFMT) before sending this message.

See Also

IE_GETFORMAT, IE_SETDRAWOPTS, **STRKFMT**

IE_SETGRIDORIGIN

Sets the origin of the rules or grid lines.

Parameters

wParam

Not used; must be 0.

lParam

The low-order word has the x-coordinate of the origin and the high-order word has the y-coordinate of the origin.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

Rules and grids are painted after the background and before the ink (and the IN_PREPAINT notification), and are treated separately from both.

The default setting is for no grid lines. The grid lines act purely as guides for the user; the ink does not interact with the grid in any way. The specification of the grid lines is in the MM_TEXT mapping mode (that is, display pixel).

See Also

IE_GETBKGND, IE_GETGRIDPEN, IE_GETGRIDSIZE, IE_SETGRIDORIGIN, IE_SETGRIDPEN, IE_SETGRIDSIZE, IE_SETBKGND

IE_SETGRIDPEN

Sets the GDI pen for the background rules or grid lines.

Parameters

wParam

Not used; must be 0.

lParam

Contains a handle to a GDI pen used to draw the grid lines.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

Rules and grids are painted after the background and before the ink (and the IN_PREPAINT notification), and are treated separately from both. A GDI pen, if specified, becomes the property of the ink edit control and must not be deleted or otherwise used by the application. If the application needs to change the settings, it must create a new pen each time it sends the IE_SETGRIDPEN message.

The default setting is for no grid lines. The grid lines act purely as guides for the user; the ink does not interact with the grid in any way. The specification of the grid lines is in the MM_TEXT mapping mode (that is, display pixel). If the GDI pen handle is NULL, the ink edit control will use a default pen. (The default pen attributes are a solid line, a width of 1 pixel, and the window grayed text color.) The maximum grid spacing is 255 pixels.

See Also

IE_GETBKGND, IE_GETGRIDORIGIN, IE_GETGRIDPEN, IE_GETGRIDSIZE, IE_SETGRIDORIGIN, IE_SETGRIDSIZE, IE_SETBKGND

IE_SETGRIDSIZE

Sets the vertical and horizontal spacing of the rules or grid lines.

Parameters

wParam

Not used; must be 0.

lParam

The low-order word has the horizontal spacing and the high-order word has the vertical spacing.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

Rules and grids are painted after the background and before the ink (and the IN_PREPAINT notification), and are treated separately from both.

The default setting is for no grid lines. The grid lines act purely as guides for the user; the ink does not interact with the grid in any way. The specification of the grid lines is in the MM_TEXT mapping mode (that is, display pixel).

See Also

IE_GETBKGND, IE_GETGRIDORIGIN, IE_GETGRIDPEN, IE_GETGRIDSIZE, IE_SETGRIDORIGIN, IE_SETGRIDPEN, IE_SETBKGND

IE_SETINK

Sets the contents of an ink edit control.

Parameters

wParam

Contains IESI_REPLACE to replace any existing control contents with the supplied ink, or IESI_APPEND to append the supplied ink to the existing contents of the ink edit control.

lParam

Contains a handle to pen data with which to initialize or reinitialize the contents of the control. If NULL, the contents of the control are cleared; any pen data in the control is discarded.

Return Value

Returns IER_OK if successful; otherwise, returns one of the following:

Constant	Description
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_MEMERR	A memory error occurred.
IER_SECURITY	The control has security protection disallowing the operation.
IER_SCALE	Attempted to merge ink of incompatible scale factors.

Comments

The application can clear or change the contents of the control at any time with this function. During the creation of controls with existing contents, this message might be sent in response to the WM_CTLINIT message.

The **HPENDATA** handle becomes the property of the control; the application must make no further use of the handle if the message returns success. On a merge operation, the original **HPENDATA** is destroyed following a successful merge. If the result of IE_SETINK indicates there is no ink left in the control, the mode is reset to IEMODE_READY if the previous mode was either IEMODE_ERASE or IEMODE_LASSO. The corresponding IN_MODECHANGED notification is also sent at this time.

See Also

IE_GETINK

IE_SETINKINPUT

Sets the ink input options for an ink edit control.

Parameters

wParam

Consists of one or two flags specifying the new ink input option. Any one of the following flags can be used as the *wParam* value. If IEI_MOVE is specified, one (and only one) additional value can combined to indicate a secondary option if all ink will not fit inside the control when moved. For example, the combi-nation IEI_MOVE|IEI_RESIZE specifies to move ink into the control and resize the control to fit if necessary. Bitwise-OR combinations of constants not including IEI_MOVE are not valid.

Constant	Description
IEI_MOVE	Move all ink inside the control.
IEI_RESIZE	Resize all ink to fit inside the control.
IEI_CROP	Crop all ink that falls outside the control.
IEI_DISCARD	Discard all ink if any falls outside the control.

lParam

Not used; must be 0.

Return Value

Returns the previous ink input bits if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

An application can dynamically modify the ink input options. If more than 1 bit is set, the order of priority is as listed in the previous table.

See Also

IE_GETINKINPUT

IE_SETMODE

Sets the control to a particular mode.

Parameters

wParam

Specifies the mode to set the control to, as follows:

Constant	Description
IEMODE_READY	The control is ready for inking, moving strokes, tapping, and so forth.
IEMODE_ERASE	The control is set to erasing mode.
IEMODE_LASSO	The control is set to lasso selection mode.

lParam

Not used; must be 0.

Return Value

Returns one of the following values:

Constant	Description
IER_OK	Success.
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_PENDATA	Attempted to set erase or lasso mode with a null **HPENDATA** handle in the control.

See Also

IE_GETMODE

IE_SETMODIFY

Sets the modify bit in the control, indicating whether the contents of the control have been modified.

Parameters

wParam

The new value of the modify bit. Must be either TRUE or FALSE.

lParam

Not used; must be 0.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The modify bit is set in the control whenever the user takes some action that changes the ink in the control. Such actions include drawing new ink, erasing, pasting, changing attributes, and moving ink. Note that calling a function to change the contents of the control also sets the modify bit.

To preserve the value of the modify bit during some modifying action, the appli-cation must first retrieve the bit's value with IE_GETMODIFY, then restore the value after completing the action.

This command succeeds regardless of the security setting.

See Also

IE_GETMODIFY

IE_SETNOTIFY

Sets the notification options for an ink edit control.

Parameters

wParam

Consists of flags specifying the kinds of notifications required:

Constant	Description
IEN_EDIT	Require notifications of editing or command events. Notifications sent: IN_CHANGE, IN_UPDATE, IN_GESTURE, IN_COMMAND.
IEN_FOCUS	Require notifications of focus events. Notifications sent: IN_SETFOCUS, IN_KILLFOCUS.
IEN_PAINT	Require notifications of painting events. Notifications sent: IN_PREPAINT, IN_POSTPAINT.
IEN_PDEVENT	Require notifications of pointing-device events (clicks and taps). Notification sent: IN_PDEVENT.
IEN_PROPERTIES	Require notifications before bringing up the properties dialog box. Notification sent: IN_PROPERTIES.
IEN_SCROLL	Require notifications of scrolling events. Notifications sent: IN_HSCROLL, IN_VSCROLL.

lParam

Not used; must be 0.

Return Value

Returns the previous notification bits if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

An application can dynamically modify the kinds of notifications and the frequency with which they are generated.

Unless otherwise specified, the parent window receives no notifications beyond the default messages sent by Windows to the parent of a child window.

See Also

IE_GETNOTIFY

IE_SETPENTIP

Sets the default ink pen tip.

Parameters

wParam

Not used; must be 0.

lParam

Address of a **PENTIP** structure.

Return Value

Returns IER_OK if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

The pen tip specified by **PENTIP** is used for all default inking in the control. Note that the new pen tip applies only to new ink entered into the control. It does not change the attributes of existing ink in the control.

See Also

PENTIP, IE_GETPENTIP

IE_SETRECOG

Sets the recognition options of the control.

Parameters

wParam

Contains bits designating the new recognition options, as listed here:

Constant	Description
IEREC_ALL	All recognition enabled.
IEREC_GESTURE	Gesture recognition enabled.
IEREC_NONE	Recognition disabled.

lParam

Not used; must be 0.

Return Value

Returns the previous recognition bits if successful; otherwise, returns IER_PARAMERR to indicate that *wParam* or *lParam* is invalid.

Comments

By default, all recognition is enabled. Currently, IEREC_GESTURE and IEREC_ALL are equivalent.

See Also

IE_GETRECOG

IE_SETSECURITY

Sets the security options of the control.

Parameters

wParam

Contains the new security bits. The high-order word is unused and must be 0. The security flags can be combined using the bitwise-OR operator.

Constant	Description
IESEC_NOCOPY	Copying not allowed.
IESEC_NOCUT	Cutting, deleting, and clearing not allowed.
IESEC_NOPASTE	Pasting disabled.
IESEC_NOUNDO	Undo not permitted.
IESEC_NOINK	Inking not allowed.
IESEC_NOERASE	Erasing not allowed.
IESEC_NOGET	The IE_GETINK message is disabled.
IESEC_NOSET	The IE_SETINK message is disabled.

lParam

Not used; must be 0.

Return Value

Returns the previous security bits if successful; otherwise, returns one of the following:

Constant	Description
IER_PARAMERR	*wParam* or *lParam* is invalid.
IER_SECURITY	The control has security protection disallowing the operation.

See Also

IE_GETSECURITY

IE_SETSEL

Sets the selection status of a particular stroke.

Parameters

wParam

Contains the zero-based index of the stroke whose selection status is to be set. A value of IX_END sets the selection status of all the strokes in the control.

lParam

lParam is TRUE to select the stroke or FALSE to remove the selection. Other values produce an IER_PARAMERR return value.

Return Value

Returns one of the following values:

Constant	Description
IER_YES	The stroke was previously selected.
IER_NO	The stroke was not previously selected.
IER_PARAMERR	*wParam* or *lParam* is invalid.

Comments

This message affects the selection status of only the specified stroke. The selection status of other strokes remains unchanged.

See Also

IE_GETSEL

IN_CHANGE

Sent after the contents of the control have been modified and repainted.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_EDIT bit has been set using the IE_SETNOTIFY message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the iedit control in the low-order word and the IN_CHANGE notification message in the high-order word.

See Also

IE_SETNOTIFY

IN_CLOSE

Sent when the control is closing and about to be destroyed.

The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the iedit control in the low-order word and the IN_CLOSE notification message in the high-order word.

Return Value

The application should return TRUE to prevent the control from being closed or FALSE for default handling.

IN_COMMAND

Sent when the user has selected an item from the pop-up menu.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_EDIT bit has been set using the IE_SETNOTIFY message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the iedit control in the low-order word and the IN_COMMAND notification message in the high-order word.

Return Value

The application should return TRUE to discard the command selection or FALSE for default processing.

Comments

The application can retrieve details about the selection by using the IE_GETCOMMAND message.

See Also

IE_GETCOMMAND, IE_SETNOTIFY

IN_ERASEBKGND

Sent to the parents of ink edit controls that have the IEB_OWNERDRAW option to request the painting of the control background.

The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_ERASEBKGND notification message in the high-order word.

Comments

The application should use the IE_GETPAINTDC message to retrieve such information as the correct device context and clipping region.

See Also

IE_GETPAINTDC

IN_GESTURE

Sent when the user has performed a gesture in the iedit control.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_EDIT bit has been set using the IE_SETNOTIFY message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_GESTURE notification message in the high-order word.

Return Value

The application should return TRUE to discard the gesture or FALSE for default processing.

Comments

An application can retrieve details about the gesture by using the IE_GETGESTURE message.

See Also

IE_SETNOTIFY

IN_HSCROLL

The IN_HSCROLL notification message is sent when the user has clicked the ink edit control's horizontal scroll bar.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_SCROLL bit has been set using the IE_SETNOTIFY message. This bit is set by default and should be cleared if the control's parent does not require this notification message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_HSCROLL notification message in the high-order word.

Return Value

The application should return TRUE to discard the scrolling request.

See Also

IE_SETNOTIFY

IN_KILLFOCUS

Sent to inform the parent window that the control is losing the focus.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_FOCUS bit has been set using the IE_SETNOTIFY message. This bit is set by default and should be cleared if the control's parent does not require this notification message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the iedit control in the low-order word and the IN_KILLFOCUS notification message in the high-order word.

Return Value

The application should return TRUE to prevent the control from losing the focus.

See Also

IE_SETNOTIFY

IN_MEMERR

Sent when the system is unable to satisfy a memory request made by the control.

The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_MEMERR notification message in the high-order word.

Return Value

The application should return TRUE to retry the operation (generally after it frees memory). If it returns TRUE and the control still cannot perform the memory operation, another IN_MEMERR notification is generated.

Comments

The ink edit control does not display an error message of any kind. Any such error messages must be displayed by the application.

IN_MODECHANGED

Sent after the control mode has changed.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_EDIT bit has been set using the IE_SETNOTIFY message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_MODECHANGED notification message in the high-order word.

See Also

IE_SETNOTIFY, IE_GETMODE, IE_SETMODE

IN_PAINT

Sent to the parent window of an owner-draw ink edit control to indicate that the control should be painted.

The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_PAINT notification message in the high-order word.

Comments

The application should use the IE_GETPAINTDC message to retrieve the details of the required painting.

See Also

IE_GETPAINTDC

IN_PDEVENT

Sent when a pointing-device transition event (such as a tap, up-click, or double-tap) occurs.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_PDEVENT bit has been set using the IE_SETNOTIFY message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_PDEVENT notification message in the high-order word.

Return Value

The application should return TRUE to discard the event, or FALSE for normal processing.

Comments

The application can retrieve a **PDEVENT** structure describing the event by sending the IE_GETPDEVENT message.

See Also

IE_GETPDEVENT, IE_SETNOTIFY

IN_POSTPAINT

Sent to inform the parent that painting is finished.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_PAINT bit has been set using the IE_SETNOTIFY message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_POSTPAINT notification message in the high-order word.

Comments

The application can send the IE_GETPAINTDC message to retrieve a device con-text with the correct clipping region to perform any additional painting on top of the ink edit control.

See Also

IE_GETPAINTDC, IE_SETNOTIFY

IN_PREPAINT

Sent just before the control paints the ink.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_PAINT bit has been set using the IE_SETNOTIFY message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_PREPAINT notification message in the high-order word.

Comments

The application can send the IE_GETPAINTDC message to retrieve a device context with the correct clipping region to perform any additional painting before the ink edit control paints.

See Also

IE_GETPAINTDC, IE_SETNOTIFY

IN_PROPERTIES

Signals the iedit control's standard Properties dialog box is about to be displayed on the screen.

The control's parent window receives the IN_PROPERTIES notification message through a WM_COMMAND message if the IEN_PROPERTIES bit has been set using the IE_SETNOTIFY message. This bit is set by default and should be cleared if the control's parent does not require this notification message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_PROPERTIES notification message in the high-order word.

Return Value

The application should return TRUE to avoid displaying the iedit control's standard Properties dialog box.

See Also

IE_SETNOTIFY

IN_SETFOCUS

Sent to inform the parent window that the control is gaining the focus.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_FOCUS bit has been set using the IE_SETNOTIFY message. This bit is set by default and should be cleared if the control's parent does not require this notification message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_SETFOCUS notification message in the high-order word.

Return Value

The application should return TRUE to prevent acquiring the focus.

See Also

IE_SETNOTIFY

IN_UPDATE

Sent when the contents of the control have been modified but not yet repainted.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_EDIT bit has been set using the IE_SETNOTIFY message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_UPDATE notification message in the high-order word.

See Also

IE_SETNOTIFY

IN_VSCROLL

Sent when the user has clicked the ink edit control's vertical scroll bar.

The control's parent window receives this notification message through a WM_COMMAND message if the IEN_SCROLL bit has been set using the IE_SETNOTIFY message. This bit is set by default and should be cleared if the control's parent does not require this notification message.

Parameters

wParam

Specifies the identifier of the ink edit control.

lParam

Specifies the handle of the ink edit control in the low-order word and the IN_VSCROLL notification message in the high-order word.

Return Value

The application should return TRUE to discard the scrolling request.

See Also

IE_SETNOTIFY

PE_BEGINDATA

Sent to the window specified by the **htrgTarget** member of the **TARGET** structure the first time any pen data is directed toward that window. Submessage of WM_PENEVENT.

Parameters

wParam

PE_BEGINDATA

lParam

Address of a **TARGET** structure.

Comments

A target window can initialize the **dwData** member of the **TARGET** structure with a handle to pen data (**HPENDATA**), a handwriting recognition object (**HRC**), or some private data type. If **dwData** is an **HPENDATA** object, the **HPENDATA** object should be in standard scale with no OEM data. For example, the **HPENDATA** object can be created as follows:

```
CreatePenDataEx(NULL, PDTS_STANDARDSCALE, CPD_TIME, 0);
```

The target window informs Windows of the type of data in the **dwData** member of the **TARGET** structure by returning LRET_HPENDATA, LRET_HRC, or LRET_PRIVATE (or LRET_DONE). If the application lets the message fall down to the Windows **DefWindowProc** function, then the function creates a handwriting recognition object (**HRC**) for this target.

The window can ignore all input except gestures. In this case, it must create a handwriting recognition object and customize it to recognize only gestures.

For an example of how an application can handle PE_BEGINDATA, see the source code for PENAPP.C in Chapter 7, "A Sample Pen Application."

See Also

TARGET, WM_PENEVENT

PE_BEGININPUT

Begins default input processing. Submessage of WM_PENEVENT.

Parameters

wParam

PE_BEGININPUT

lParam

The high-order word is the handle of the window in which the pen first touched down, and the low-order word is the event reference identifier returned from **GetMessageExtraInfo**.

Return Value

Returns PCMR_OK if successful; otherwise, it returns one of the following values:

Constant	Description
PCMR_ALREADYCOLLECTING	**StartPenInput** has already been called for this session.
PCMR_APPTERMINATED	The application terminated input.
PCMR_ERROR	Parameter or unspecified error.
PCMR_INVALID_PACKETID	A packet identifier is invalid.
PCMR_SELECT	Press-and-hold action was detected. Collection is not started.
PCMR_TAP	A tap was detected. Collection is not started.

Comments

A control can initiate default pen-input processing by sending this message to its parent. If the parent allows the message to be processed by passing it to **DefWindowProc**, default pen-input behavior begins. The **DoDefaultPenInput** function uses this technique.

For further information about default processing, refer to Chapter 2, "Starting Out with System Defaults."

See Also

DoDefaultPenInput, WM_PENEVENT

PE_BUFFERWARNING

Generated by the pen driver component of the system when the input queue is approximately half full. When an application receives PE_BUFFERWARNING, it should immediately call **GetPenInput** to drain the input queue. Submessage of WM_PENEVENT.

Parameters

wParam

PE_BUFFERWARNING.

lParam

Extra information encapsulating a reference to the event and the **HPCM** that generated it. Applications can use the **EventRefFromWpLp** and **HpcmFromWpLp** macros to retrieve these values.

Comments

If an application receives this message, it has fallen behind in processing the input. The buffer is in danger of overflowing. The application should repeatedly call **GetPenInput** to gather the unprocessed pen input.

See Also

GetPenInput, WM_PENEVENT

PE_ENDDATA

Sent at the termination of pen input to all windows specified by the **htrgTarget** member of the **TARGET** structure that have received a PE_BEGINDATA message. Submessage of WM_PENEVENT.

Parameters

wParam

PE_ENDDATA.

lParam

Address of a **TARGET** structure.

Comments

A target can inform the recognizer that pen input has ended and process the **HRC** in response to this message. If this message is allowed to fall through to the **DefWindowProc** function, default result processing is done if the value in the **dwData** member of the **TARGET** structure is of type **HRC**. **EndPenInput** is called followed by a call to **ProcessHRC**. Then the PE_RESULT submessage is sent to the target to allow the target to get results and process them.

If the **dwData** member of the **TARGET** structure has the **HPENDATA** or **HRC** type, the object is destroyed by **DefWindowProc** on completion of results processing.

See Also

TARGET, WM_PENEVENT

PE_ENDINPUT

Sent to indicate that the default collection has terminated. Submessage of WM_PENEVENT.

Parameters

wParam

PE_ENDINPUT.

lParam

Not used.

See Also

WM_PENEVENT

PE_GETINKINGINFO

Fills an **INKINGINFO** structure. Submessage of WM_PENEVENT.

Parameters

wParam

PE_GETINKINGINFO.

lParam

Address of an **INKINGINFO** structure, which is initialized with default values.

Return Value

The targeted windows in the application should return 1 to customize inking information. A return value of 0 results in default inking behavior.

Comments

Before beginning default pen input, Windows sends PE_GETINKINGINFO to all the windows specified by the **htrgTarget** member of the **TARGET** structure. (The **TARGET** structure is part of the **TARGINFO** structure created during processing of the earlier PE_SETTARGETS messages.)

The default values are the same as those used when **StartInking** is called with *lpinkinginfo* set to NULL, but the PII_SAVEBACKGROUND flag is forced on in the **wFlags** member of the **INKINGINFO** structure to automatically save and restore the inking background.

The **hrgnClip** member of the **INKINGINFO** structure temporarily contains the index of the target in the **TARGINFO** structure retrieved by the PE_SETTARGETS message. Note that this is an overloading of this member to identify the targets. If the **htrgTarget** window specified in the **TARGET** structure returns 1 to this message, the following actions are taken:

- If both the PII_INKPENTIP and PII_RECTCLIP flags are set, the pen tip specified by the **tip** member of the **INKINGINFO** structure is saved and used whenever the pen goes down within the area defined by the **rectClip** member. In most cases, the ink color changes at or near the clipping boundary, even when the pen is dragged over it. Because inking is done on a per-segment basis, there may be a slight overlap of color near a common boundary.
- If the width given in the **tip** member is 0, no ink will appear within the area specified by the **rectClip** member. Password fields can be implemented using this technique.
- If the PII_SAVEBACKGROUND flag is clear (0), any ink dropped within the area specified by the **rectClip** member is not removed at the termination of the collection. The ink edit control, for example, uses this technique. However, the parent window can override this default behavior when it finally receives a PE_GETINKINGINFO message after all its targets have been called.
- If the PII_INKSTOP flag is set, the **rectInkStop** member is folded into the region specified by the **hrgnInkStop** member, which is used in calls to the **StartInking** function. When inking stops due to a pen-down event in the **rectInkStop** rectangle, a WM_PENMISC message with the PMSC_INKSTOP submessage is sent to the window specified by the **htrgTarget** member of the **TARGET** structure. *lParam* is the same as in the PE_PENDOWN message that caused the inking to stop. As with PII_SAVEBACKGROUND, *hwnd* can override the preprocessed values accumulated by the targets.

For further information about PE_GETINKINGINFO, see Chapter 2, "Starting Out with System Defaults."

See Also

INKINGINFO, PE_SETTARGETS, **StartInking**, **TARGET**, **TARGINFO**, WM_PENEVENT

PE_GETPCMINFO

Fills a **PCMINFO** structure, which is then used in a call to **StartPenInput**. Submessage of WM_PENEVENT.

Parameters

wParam

PE_GETPCMINFO.

lParam

Address of a **PCMINFO** structure, which is initialized with default values.

Comments

The default values are the same as those applied when **StartPenInput** is called with the *lppcminfo* parameter set to NULL:

- Inking terminates when the time-out period elapses.
- Inking terminates when a tap occurs outside the client rectangle of *hwnd*.
- Inking does not start if the initial pen input consists of a press-and-hold gesture.

However, an exclusion region specified by the **hrgnExclude** member of the **PCMINFO** structure may have accumulated while processing PE_SETTARGETS messages. The window procedure can modify the values to customize the collection parameters before pen input begins.

If the PCM_DOPOLLING flag in the **dwPcm** member of the **PCMINFO** structure is set, it is disregarded and pen input remains in event mode.

See Also

PCMINFO, PE_SETTARGETS, **StartPenInput**, WM_PENEVENT

PE_MOREDATA

Sent to the window specified by the **htrgTarget** member of the **TARGET** structure to indicate that more pen data is available for that window. Submessage of WM_PENEVENT.

Parameters

wParam

PE_MOREDATA.

lParam

Address of an **INPPARAMS** structure.

Comments

DefWindowProc collects the pen input in response to the PE_PENDOWN, PE_PENUP, and PE_PENMOVE messages and sends the input on a stroke-by-stroke basis to one of the targets in the **TARGINFO** structure. On a pen-tip transition—that is, from pen-down to pen-up or vice versa — the Windows **DefWindowProc** function sends a PE_MOREDATA message to the window specified by the **htrgTarget** member of the **TARGET** structure identified in the PE_BEGINDATA message.

If it receives a PE_MOREDATA message, **DefWindowProc** uses **AddPointsPendata** or **AddPenInputHRC**, or does nothing, depending on whether the data type in the **dwData** member of the **TARGET** structure is a handle to an **HPENDATA**, an **HRC** handle for handwriting recognition, or some private data type.

See Also

AddPenInputHRC, **AddPointsPendata**, **INPPARAMS**, PE_PENMOVE, PE_PENDOWN, PE_PENUP, **TARGET**, **TARGINFO**, WM_PENEVENT

PE_PENDOWN

Generated by the pen driver component of the system when the pen tip touches the tablet surface. Submessage of WM_PENEVENT.

Parameters

wParam

PE_PENDOWN.

lParam

Extra information encapsulating a reference to the event and the **HPCM** that generated it. Applications can use the **EventRefFromWpLp** and **HpcmFromWpLp** macros to retrieve these values.

See Also

WM_PENEVENT

PE_PENMOVE

Generated by the pen driver component of the system when the pen moves, forcing more packets into the input queue. Submessage of WM_PENEVENT.

Parameters

wParam

PE_PENMOVE.

lParam

Extra information encapsulating a reference to the event and the **HPCM** that generated it. Applications can use the **EventRefFromWpLp** and **HpcmFromWpLp** macros to retrieve these values.

Comments

This message is analogous to WM_MOUSEMOVE. It provides notification that the pen is moving. Like its mouse counterpart, PE_PENMOVE messages are coalesced so that only a single such message exists in the application's message queue. How-ever, the event the message represents is the first of these coalesced events, not the last event, as is the case with WM_MOUSEMOVE.

An application need not handle this message if transition events like PE_PENUP and PE_PENDOWN are sufficient notification. PE_PENMOVE is useful when an application must monitor pen movement with greater frequency than PE_PENUP or PE_PENDOWN allows.

See Also

WM_PENEVENT

PE_PENUP

Generated by the pen driver component of the system when the pen tip leaves the tablet surface. Submessage of WM_PENEVENT.

Parameters

wParam

PE_PENUP.

lParam

Extra information encapsulating a reference to the event and the **HPCM** that generated it. Applications can use the **EventRefFromWpLp** and **HpcmFromWpLp** macros to retrieve these values.

See Also

WM_PENEVENT

PE_RESULT

During the processing of the PE_ENDDATA submessage, sent to all windows specified by the **htrgTarget** member of the **TARGET** structure that have received a PE_BEGINDATA message. PE_RESULT applies only to recognition and is sent only if the value in the **dwData** member of the **TARGET** structure is of type **HRC**. Submessage of WM_PENEVENT.

Parameters

wParam

PE_RESULT.

lParam

HRC object for the recognition session.

Comments

A target can perform recognition-result processing in response to PE_RESULT. If it receives this message, the Windows **DefWindowProc** function performs default result processing. The first **HRCRESULT** object for the **HRC** is obtained using the **GetResultsHRC** function. The **HRCRESULT** handle is used to retrieve a string of symbols that are converted, one by one, into characters. The characters are then posted as WM_CHAR messages to the window specified by the **htrgTarget** member of the **TARGET** structure.

The clear, cut, copy, paste, and undo gestures are converted to WM_CLEAR, WM_CUT, WM_COPY, WM_PASTE, and WM_UNDO messages. They are posted to the **htrgTarget** target, together with appropriate mouse messages, so that the target can perform appropriate processing (such as selection) before applying the gestures.

See Also

TARGET, WM_PENEVENT

PE_SETTARGETS

Sent to an application window so that it can set its own targeting structure. Submessage of WM_PENEVENT.

Parameters

wParam

PE_SETTARGETS.

lParam

Address of a far pointer to a **TARGINFO** structure. (Note that *lParam* is a pointer to a pointer.)

Return Value

The application should return LRET_DONE to indicate that it has set up the targeting information for the child windows. A return of 0 indicates that the application is the only target. The application can also return LRET_ABORT to abort the targeting process altogether.

Comments

The **DefWindowProc** function enumerates all the descendants of the window specified in its first parameter and sends each one a PE_GETPCMINFO message. For every descendant that returns 1 to this message, the PCM_RECTBOUND and PCM_RECTEXCLUDE flags of the **PCMINFO** structure are examined. If the PCM_RECTBOUND flag is set, the descendant is included in the list of potential targets and the **rectBound** member in **PCMINFO** is copied to the **rectBound** member of the **TARGET** structure. If the PCM_RECTEXCLUDE flag is set, the **rectExclude** member of **PCMINFO** is added to an exclusion region that is passed (as the **hrgnExclude** member of the **PCMINFO** structure) to the **StartInput** call. If there are no descendants, or if the window procedure of *hwnd* returns 0, a **TARGINFO** structure is constructed with *hwnd* as the single target.

For default processing behavior, the application should allow PE_SETTARGETS to fall through to **DefWindowProc**. A PE_GETPCMINFO message will follow to establish targets or termination conditions (buttons, for example).

For further information about PE_SETTARGETS, see Chapter 2, "Starting Out with System Defaults."

An application can replace the default targeting with a set of targets it defines itself. In this case, the application allocates enough memory for the **TARGINFO** structure plus all the **TARGET** structures.

Example

The following example illustrates how to handle PW_SETTARGETS for *n* targets, where each target is in the array *rgHwnd*. Notice the code increases the allocation by *n*-1 **TARGET** structures, since **TARGINFO** already contains one **TARGET** structure.

```
DWORD cbAlloc         = sizeof(TARGINFO) + (n-1) * sizeof(TARGET);
HGLOBAL hTargets      = GlobalAlloc( GHND, cbAlloc );
LPTARGINFO lptarginfo = GlobalLock( hTargets );

lptarginfo->cbSize    = sizeof(TARGINFO);
lptarginfo->cTargets  = n;                    // Number of targets
lptarginfo->htrgOwner = HtrgFromHwnd(hwnd);   // Macro in penwin.h
lptarginfo->dwFlags   = TPT_TEXTUAL;          // For text

for (i = 0; i < n; i++)
{
     HWND hwnd = (HTRG)rgHwnd[i];              // Window of this target
```

```
      lptarginfo->rgTarget[i].dwFlags    = 0;   // Reserved
      lptarginfo->rgTarget[i].idTarget   = i;
      lptarginfo->rgTarget[i].htrgTarget = HtrgFromHwnd(hwnd);
      lptarginfo->rgTarget[i].dwData = 0;

      // Use screen coords of each window:
   {
// Note: rectBound is a RECTL. In 16-bit code, one has to assign each
// field separately. In 32-bit code, you can use the rectBound directly.
   RECT rect;
      GetClientRect( hwnd, &rect);
      ClientToScreen( hwnd, (LPPOINT)&rectBound.left);
      ClientToScreen( hwnd, (LPPOINT)&rectBound.right);
   lptarginfo->rgTarget[i].rectBound.left = rect.left;
   lptarginfo->rgTarget[i].rectBound.top = rect.top;
   lptarginfo->rgTarget[i].rectBound.right = rect.right;
   lptarginfo->rgTarget[i].rectBound.bottom = rect.bottom;
   }
}

// Return our structures:
*(LPTARGINFO FAR *)lParam = lptarginfo;
```

See Also

PCMINFO, TARGET, TARGINFO, WM_PENEVENT

PE_TERMINATED

Generated by the pen driver component of the system when pen input terminates. Submessage of WM_PENEVENT.

Parameters

wParam

PE_TERMINATED.

lParam

Extra information encapsulating a reason for termination and the current **HPCM**. Applications can use the **TerminationFromWpLp** and **HpcmFromWpLp** macros to retrieve these values.

Comments

When an application receives the PE_TERMINATED message, collection has already terminated and the **HPCM** handle returned from **StartPenInput** has become invalid. PE_TERMINATED indicates an application should perform such tasks as final results processing, repainting, and cleanup.

See Also

WM_PENEVENT

PE_TERMINATING

Generated by the pen driver component of the system when pen input is about to terminate. Submessage of WM_PENEVENT.

Parameters

wParam

PE_TERMINATING.

lParam

Extra information encapsulating a reason for termination and the current **HPCM**. Applications can use the **TerminationFromWpLp** and **HpcmFromWpLp** macros to retrieve these values.

Comments

When it receives PE_TERMINATING, the application must immediately retrieve any remaining points.

See Also

WM_PENEVENT

WM_CTLINIT

Sent to the parent of a bedit, hedit, or iedit control while the control is being created in order to get extra information about the control.

Parameters

wParam

Type of control. This parameter can be CTLINIT_BEDIT, CTLINIT_IEDIT, or CTLINIT_HEDIT.

lParam

Address of a control structure, depending on *wParam*. For values of CTLINIT_BEDIT, CTLINIT_HEDIT, or CTLINIT_IEDIT in *wParam*, *lParam* points to either a **CTLINITBEDIT**, **CTLINITHEDIT**, or **CTLINITIEDIT** structure, respectively.

Comments

Each of the CTLINIT structures has its first three members already initialized: **cbSize** (size of the structure), **hwnd** (handle to the control window), and **id** (child identifier of the control). The parent of the control can set appropriate values to the rest of the members in the structure and the control will then use those values when initializing itself.

See Also

CTLINITHEDIT, **CTLINITBEDIT**, **CTLINITIEDIT**

WM_GLOBALRCCHANGE

See WM_PENMISCINFO.

WM_HOOKRCRESULT

Sent to a window before WM_RCRESULT is sent to the target window.

The WM_HOOKRCRESULT message is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

Parameters

wParam

REC_ code indicating why recognition ended.

lParam

Address of an **RCRESULT** structure.

Comments

The application may examine the results in the **RCRESULT** structure. Changing any of the values leads to unpredictable results. The application should make a copy of any information it needs.

See Also

RCRESULT, **SetRecogHook**, WM_RCRESULT

WM_PENCTL

Performs several actions, including:

- Converts a logical character position to a byte offset.
- Converts a byte offset to a logical character position.
- Switches the font in a bedit control to the default font.

Parameters

wParam

Submessage identifier as described in the following table. Each submessage is documented separately.

Constant	Description
HE_CANCELCONVERT	Cancels Kana-to-Kanji conversion. (Japanese version only.)
HE_CHAROFFSET	Converts logical character position of a character in the control to byte offset to the character. For bedit controls only.
HE_CHARPOSITION	Converts byte offset in the text buffer of the control to the logical character position, which contains the byte specified by the byte offset. For bedit controls only.
HE_DEFAULTFONT	Switches the font of the bedit control to the default font that the bedit selects at the time of creation. For bedit controls only.
HE_ENABLEALTLIST	Enables or disables the alternate list in a bedit control.
HE_FIXKKCONVERT	Confirm undetermined string and close Input Method Editor (IME). (Japanese version only.)
HE_GETBOXLAYOUT	Points to the **BOXLAYOUT** structure, which is filled with the current box layout for the control. For bedit controls only.
HE_GETCONVERTRANGE	Gets the range of the marked conversion string. (Japanese version only.)
HE_GETINFLATE	**LPRECTOFS** filled with current value.
HE_GETINKHANDLE	Retrieves a handle to the captured ink.
HE_GETKKCONVERT	Determines if the Input Method Editor (IME) is in pen (or keyboard) conversion mode. (Japanese version only.)
HE_GETKKSTATUS	Determines the mode of the Kana-to-Kanji conversion. (Japanese version only.)

HE_GETRC	Fills an **RC** structure, whose address is passed in the *lParam*, with current values. See the note that follows.
HE_GETUNDERLINE	Queries whether underline mode is set. For hedit controls only.
HE_HIDEALTLIST	Hides the alternate list in a bedit control, assuming it is being displayed.
HE_KKCONVERT	Starts Kana-to-Kanji conversion. (Japanese version only.)
HE_PUTCONVERTCHAR	Sends a character, marked for conversion, to the IME. (Japanese version only.)
HE_SETBOXLAYOUT	Sets a **BOXLAYOUT** structure. For bedit controls only.
HE_SETCONVERTRANGE	Sets the range of the marked conversion string. (Japanese version only.)
HE_SETINFLATE	Specifies adjustments to the control window to specify the size of the writing window.
HE_SETINKMODE	Starts the collection of inking.
HE_SETRC	Sets the **RC** structure, whose address is passed in the *lParam*. See the note that follows.
HE_SETUNDERLINE	Sets or cancels underline mode. For hedit controls only.
HE_SHOWALTLIST	Displays the alternate list menu in a bedit control, assuming that alternate lists are enable.
HE_STOPINKMODE	Stops the collection of ink.

The HE_GETRC and HE_SETRC submessages are provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

---

lParam

Depends on *wParam*. See the individual HE_ submessage descriptions for more information.

Comments

Any control message (a message with the EM_ prefix) that can be sent to an edit control can also be sent to an hedit window. Most of these control messages are also supported by bedit controls.

The HE_ submessages are also common to both hedit and bedit controls except as noted in the preceding table. In a bedit control, each cell contains one logical character. Carriage return (CR) and line-feed (LF) bytes together form one logical character.

Before using the HE_SETBOXLAYOUT or HE_SETINFLATE submessages, it is often useful to retrieve the current structure associated with the control using the HE_GETBOXLAYOUT or HE_GETINFLATE submessages. You should then change the appropriate members in the retrieved structure. This procedure reduces the risk of inadvertent changes to the structure.

In older applications compatible with version 1.0 of the Pen API, placing the value RRM_SYMBOL in

wResultMode of the **RC** structure disables all default diction-ary processing in a bedit control. The 1.0 application can perform dictionary pro-cessing on its own by retrieving the recognition results during the processing of the HN_RESULT notification and calling the **DictionarySearch** function.

See Also

HE_CANCELCONVERT, HE_CHAROFFSET, HE_CHARPOSITION, HE_DEFAULTFONT, HE_ENABLEALTLIST, HE_FIXKKCONVERT, HE_GETBOXLAYOUT, HE_GETIMEDEFAULT, HE_GETINFLATE, HE_GETINKHANDLE, HE_GETKKCONVERT, HE_GETKKSTATUS, HE_GETUNDERLINE, HE_SETBOXLAYOUT, HE_SETIMEDEFAULT, HE_SETINFLATE, HE_SETINKMODE, HE_SETUNDERLINE, HE_SHOWALTLIST, HE_STOPINKMODE

WM_PENEVENT

Sent to an application after **StartPenInput** has initiated a pen collection.

Parameters

wParam

Submessage identifier as described in the following table. Each submessage is documented separately.

Constant	Description
PE_BEGINDATA	Initialization message to all targets.
PE_BEGININPUT	Begin default input.
PE_BUFFERWARNING	The input queue is getting full. The application should call **GetPenInput**.
PE_ENDDATA	Termination message to all targets.
PE_RESULT	Recognition result message to all targets.
PE_ENDINPUT	Input termination message to window.
PE_GETINKINGINFO	Get inking information.
PE_GETPCMINFO	Get input collection information.
PE_MOREDATA	Target gets more data.
PE_PENMOVE	The pen moved, placing more packets into the input queue without a tip transition. This mes-sage is coalesced with other PE_PENMOVE messages, so the Windows queue has only a single such message waiting.
PE_PENDOWN	The pen tip went down.
PE_PENUP	The pen tip went up.
PE_SETTARGETS	Set **TARGINFO** target data structure.
PE_TERMINATED	Pen input terminated. The **HPCM** handle for the current collection has become invalid.
PE_TERMINATING	Pen input is about to terminate. The application must retrieve any remaining points immediately.

lParam

Depends on *wParam*. In most cases, this is extra information encapsulating a reference to the event and the **HPCM** that generated it. These are retrieved using the **EventRefFromWpLp** and **HpcmFromWpLp** macros.

Comments

This message is not sent if polling is used—that is, if the **dwPcm** member of the **PCMINFO** structure contains the PCM_DOPOLLING flag.

See Also

GetPenInput, **StartPenInput**, **TARGINFO**

WM_PENMISC

Sent to notify an application of some pen-related change, such as a change in a bedit control. WM_PENMISC is also used to get information from a window about pen-related attributes.

Parameters

wParam

One of the following subfunctions:

PMSC_BEDITCHANGE

Indicates that system settings for bedit controls have been changed. When it receives this message, a bedit control updates its state according to the settings indicated by the **BOXEDITINFO** structure that *lParam* points to.

PMSC_GETHRC

Return a copy of the **HRC** handle associated with the window. *lParam* is unused and should be set to 0. If a window has no associated **HRC** structure, NULL is returned. It is the caller's responsibility to destroy any **HRC** the message returns.

PMSC_GETINKINGINFO

Retrieve the **INKINGINFO** structure associated with the window and copy it to the structure pointed to by *lParam*. The message is ignored if *lParam* is NULL.

PMSC_GETPCMINFO

Retrieve the **PCMINFO** structure associated with the window and copy it to the structure pointed to by *lParam*. If a window has no associated **PCMINFO** structure, NULL is returned. The message is ignored if *lParam* is NULL. The system initializes **PCMINFO** as follows:

- **dwPcm** is a combination of the PCM_RECTBOUND, PCM_TIMEOUT, and PCM_TAPNHOLD flags.
- **rectBound** is the client area of *hwnd*, in screen coordinates.
- **dwTimeout** is the current writing time-out in milliseconds, as reported by **GetPenMiscInfo** using PMI_TIMEOUT.
- All other members are 0.

PMSC_INKSTOP

Inking has stopped because of a pen-down event. *lParam* contains the **HPCM** handle corresponding to the collection and the event reference at which the inking stopped. An application can retrieve these values with the **HpcmFromWpLp** and **EventRefFromWpLp** macros, respectively.

PMSC_KKCTLENABLE

WM_PENMISC is broadcast when kana-kanji controls are enabled. (Japanese version only.)

PMSC_LOADPW

WM_PENMISC is broadcast when PENWIN.DLL loads or unloads. *lParam* is one of the following:

- PMSCL_LOADED (PENWIN.DLL just loaded).
- PMSCL_UNLOADED (PENWIN.DLL just unloaded).
- PMSCL_UNLOADING (PENWIN.DLL is about to unload).

PMSC_PENUICHANGE

Broadcast to indicate that the pen user interface DLL (PENUI) has been changed. (Japanese version only.)

PMSC_SETHRC

Associate the **HRC** handle in *lParam* with the window. The window makes a copy of the **HRC** for itself so that the sender of the message can destroy its copy. Returns nonzero if successful; otherwise, returns 0.

PMSC_SETINKINGINFO

Associate the **INKINGINFO** structure pointed to by *lParam* with the window. Returns nonzero if successful; otherwise, returns 0.

PMSC_SETPCMINFO

Associate the **PCMINFO** structure pointed to by *lParam* with the window. The **cbSize** member of the structure must be initialized with sizeof(PCMINFO). Returns nonzero if successful; otherwise, returns 0.

PMSC_SUBINPCHANGE

Indicates the character finder DLL (SUBINPUT) has been changed. (Japanese version only.)

PMSC_GETSYMBOLCOUNT

Retrieve the number of symbols contained in the last recognition result. *lParam* should be 0. This message should be sent by the window that received the HN_RESULT notification before returning from the notification message.

PMSC_GETSYMBOLS

Retrieve the symbols contained in the last recognition result. *lParam* should be a pointer to a buffer large enough to accommodate the number of symbols contained in the result followed by SYV_NULL. The number of symbols in the result can be obtained by sending the WM_PENMISC message to the window with the PMSC_GETSYMBOLCOUNT submessage. This message should be sent by the window that received the HN_RESULT notification before returning from the notification message. A nonzero value is returned to indicate success.

PMSC_SETSYMBOLS

Change the symbols for the last recognition result. *lParam* should be a pointer to a buffer containing the array of symbols to be set terminated by SYV_NULL. *lParam* may be NULL to indicate an empty result. The control receiving this message should not perform any garbage detection on results set in this manner. This allows the application to perform its own garbage detection. In the case of the bedits controls, the number of symbols set must be the same as the number of symbols obtained using the WM_PENMISC message with the PMSC_GETSYMBOLCOUNT submessage. If not, the symbols are not set. This message should be sent by the window that received the HN_RESULT notification before returning from the notification message. A nonzero value is returned to indicate success.

lParam

Depends on *wParam*.

See Also

BOXEDITINFO, **INKINGINFO**, **PCMINFO**

WM_PENMISCINFO

Posted to all top-level windows whenever a pen system change is made.

Parameters

wParam

PMI_ value that identifies the system change.

lParam

New value, depending on *wParam*.

Comments

This message is broadcast to all top-level windows whenever a new global pen default is set by a call to the **SetPenMiscInfo** function. (Control Panel is the principal source of these changes.) A series of WM_PENMISCINFO broadcasts is typically made after a Control Panel application is closed.

In version 2.0 of the Pen API, the message parameters have been defined and the name of this message has been changed from WM_GLOBALRCCHANGE to WM_PENMISCINFO, although the value is the same for compatibility with version 1.0. The *wParam* and *lParam* parameters are the same as the parameter values provided to the **SetPenMiscInfo** function.

For version 1.0 compatibility, a call to the **SetGlobalRC** function also causes a posted broadcast of this message to all top-level windows, but the parameters are both 0. PMI_RCCHANGE may be used as an alias for 0 for *wParam*; however, PMI_RCCHANGE is not a valid parameter to **GetPenMiscInfo** or **SetPenMiscInfo**.

See Also

SetPenMiscInfo, **GetPenMiscInfo**, **SetGlobalRC**, PMI_

WM_RCRESULT

Sent to an application by a recognizer with the results of a recognition.

The WM_RCRESULT message is provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

Parameters

wParam

REC_ code indicating why recognition ended.

lParam

Address of an **RCRESULT** structure.

See Also

ProcessWriting, **RCRESULT**, **Recognize**, **RecognizeData**

Pen Application Programming Interface Constants

This chapter describes some of the many manifest constants defined by the Pen Application Programming Interface, listed alphabetically. Constants listed in this chapter are primarily those that do not contain a complete listing in any individual function or message description elsewhere in this documentation. Constants that pertain to individual functions or messages can be found with the descriptions of those functions or messages.

Each entry includes a complete description of the constant. For a comprehensive list of the Pen API constants, see Chapter 9, "Summary of the Pen Application Programming Interface," or refer to the PENWIN.H header file. Refer to the index to locate the descriptions of any of the Pen API constants.

ALC_ Alphabet Codes

The ALC_ constants enable a subset of the active character set, depending on the current language.

For example, the French language includes "h" in the lowercase alphabet. In the same way, "£" replaces "$" if ALC_MONETARY is set in British systems. For more information about alphabets, see "Specifying an Alphabet Set" in Chapter 8 and "Alphabet" in Chapter 5.

Comments

The following ALC_ constants are are supported:

Constant	Description
ALC_ALL	All characters except Japanese characters.
ALC_ALPHA	ALC_LCALPHA | ALC_UCALPHA.
ALC_ALPHANUMERIC	ALC_LCALPHA | ALC_UCALPHA | ALC_NUMERIC.
ALC_ASCII	Seven-bit characters ASCII #20-ASCII #0x7F
ALC_DBCS	Allow double-byte character set (DBCS) variety of single-byte character set (SBCS).
ALC_DEFAULT	Default value; uses complete set of recognizable char-acters and gestures. The set of these is defined by the recognizer. It is the set of characters at or above ALC_SYSMINIMUM that the recognizer can accurately distinguish. If an application sets ALC_DEFAULT in the **HRC** object, and the recognizer is an alphanumeric system recognizer, the recognizer must at least support ALC_SYSMINIMUM as a default. ALC_DEFAULT should be the same character set as the complete character set for the given language minus the ALC_OTHER characters. If an application combines ALC_DEFAULT with other ALC_ values, ALC_DEFAULT is ignored.
ALC_GESTURE	Gestures.
ALC_GLOBALPRIORITY	Specifies that the global recognition priorities (from Tool Palette) are to be used during recognition. An application can control its own recognition priority in a control by clearing this flag and then setting its own priorities in the HRC.
ALC_HIRAGANA	Hiragana characters. (Japanese version only.)
ALC_JIS1	All Kanji Shift JIS level 1 characters. (Japanese version only.)
ALC_KANJI	Kanji characters, Shift JIS levels 1, 2, and 3. (Japanese version only.)
ALC_KANJIALL	ALC_ALL | ALC_HIRAGANA |

	ALC_KATAKANA | ALC_KANJI. (Japanese version only.)
ALC_KANJISYSMINIMUM	Minimum set of characters needed for Japanese system recognizer. Same as ALC_SYSMINIMUM | ALC_HIRAGANA | ALC_KATAKANA | ALC_JIS1. (Japanese version only.)
ALC_KATAKANA	Katakana characters. (Japanese version only.)
ALC_LCALPHA	Lowercase letters a-z.
ALC_MATH	Math symbols: %^*()-+={}<>,/.
ALC_MONETARY	Monetary symbols: ,.$ or appropriate currency desig-nation such as the yen or pound sterling symbol, accord-ing to the current language setting.
ALC_NONPRINT	Space, tab, carriage-return, and control glyphs.
ALC_NOPRIORITY	No priority. This value means the application has no preference for one type of symbol over another.
ALC_NUMERIC	Numerals 0-9.
ALC_OEM	Bits reserved for recognizer capabilities specific to the original equipment manufacturer (OEM).
ALC_OTHER	Other symbols: @ # | _ ~ [ ]. That is, all other symbols not included in ALC_ALPHANUMERIC, ALC_MONETARY, ALC_MATH, and ALC_PUNC.
ALC_PUNC	Punctuation: !-;`"?()&.,\.
ALC_RESERVED	Reserved.
ALC_SYSMINIMUM	Minimum set of characters needed for Roman alphabet system recognizers: ALPHANUMERIC | ALC_PUNC | ALC_WHITE | ALC_GEST.
ALC_UCALPHA	Uppercase letters A-Z.
ALC_USEBITMAP	(Description follows table.)
ALC_WHITE	White space. If this value is not set in the **HRC** object, the recognizer should ignore any white space left between characters. Thus, ALC_WHITE is included in the ALC_DEFAULT. For example, in the zip code field of the Hform sample application, where ALC_NUMERIC | ALC_GESTURE is set, the user does not have to worry about getting any extraneous spaces.

If ALC_USEBITMAP is set, it indicates the recognizer should adopt an alphabet set defined by the application. The defined set specifies individual characters of an alphabet by setting bits in a 256-bit bitfield. The lowest bit corresponds to the first character of the alphabet, the second bit to the second character, and so forth.

An application passes the bitfield to a recognizer through the **SetAlphabetHRC** or **SetBoxAlphabetHRC**

functions. The following code shows how. Assume the array rgbfSet holds the desired bit values.

```
HRC     hrc;                          // HRC handle
BYTE    rgbfSet[cbRcrgbfAlcMax]       // 256-bit bitfield

SetAlphabetHRC( hrc, ALC_USEBITMAP, (LPBYTE) rgbSet );
```

ALC_USEBITMAP can be combined with other ALC_ values using the bitwise-OR operator. An application can thus, for example, select certain letters with a defined bitmap and combine them with all numerals and punctuation.

For Asian languages other than Japanese, refer to the appropriate subsets within the language: phonetic symbols for words within the language, phonetic symbols for words outside the language, and native pictographs. For example, in Korean, ALC_HANGUEL equals ALC_KATAKANA, and ALC_HANJA equals ALC_KANJI.

For kanji and other Asian encodings, different effects are possible depending on the state of ALC_DBCS. These effects are described in the following table.

Character in	**ALC_DBCS = 0**	**ALC_DBCS = 1**
ALC_HIRAGANA	N/A	Shift JIS characters 0x8154, 0x8155, and 0x829F - 0x82F1.
ALC_JIS1	N/A	All Kanji Shift JIS level 1 characters.
ALC_KATAKANA	0xA1 - 0xDF	Shift JIS characters 0x814A, 0x814B, 0x8152, 0x8153, 0x815B, and 0x8340 - 0x8396.
ALC_KANJI	N/A	All Kanji characters, Shift JIS levels 1, 2, and 3.

The following table shows the characters in Shift-JIS in each ALC_ set supported in the Japanese version:

ALC_ value	**Shift JIS Code**
ALC_HIRAGANA	0x8154, 0x8155, and 0x829F - 0x82F1
ALC_JIS1	0x8156 - 0x815A, 0x889F - 0x9872
ALC_KANJI	0x8156 - 0x815A, 0x889F - 0xEAA4, 0xED40 - 0xEDFC, 0xEE40 - 0xEEFC, 0xF040 - 0xF9FC, 0xFAF0 - 0xFAFC, 0xFB40 - 0xFBFC, 0xFC40 - 0xFC4B
ALC_KATAKANA	0x814A, 0x814B, 0x8152, 0x8153, 0x815B, 0x8340 - 0x8396
ALC_LCALPHA	0x8281 - 0x829A
ALC_MATH	0x8143, 0x8144, 0x814F, 0x815E, 0x8169, 0x816A, 0x816F, 0x8170, 0x817B - 0x817E, 0x8180 - 0x8188, 0x8193, 0x8196
ALC_MONETARY	0x8143, 0x8144, 0x818F - 0x8192
ALC_NONPRINT	0x8140
ALC_NUMERIC	0x824F - 0x8258

ALC_OTHER	0x814C - 0x814E, 0x8150, 0x8151, 0x8160 - 0x8164, 0x816B, 0x816C, 0x8171 - 0x8174, 0x8179, 0x817A, 0x817F, 0x8189 - 0x818E, 0x8194, 0x8197 - 0x81FC, 0x8240 - 0x824E, 0x8259 - 0x825F, 0x827A - 0x8280, 0x829B - 0x829E, 0x82F2 - 0x82FC, 0x837F, 0x897 - 0x83FC, 0x8840 - 0x84FC, 0x8740 - 0x879D
ALC_PUNC	0x8141 - 0x8149, 0x815B - 0x815F, 0x8165 - 0x816A, 0x816D - 0x8170, 0x8175 - 0x8178, 0x817C, 0x8195
ALC_UCALPHA	0x8260 - 0x8279
ALC_WHITE	0x8140

A recognizer must not return a symbol value outside the specified subset. However, a recognizer does not have to force a match to the subset; it can instead return "unknown" if a suitable match is not found.

You can set the ALC_ value for an hedit or bedit control in a dialog box by insert-ing a special string in the .RC file's CONTROL statement. This string is in the form ALC<*xxxx*> where *xxxx* represents a case-independent hexadecimal ALC_ code, without a preceding 0x qualifier. You can append normal window text after the ALC_ entry.

The following line demonstrates setting the ALC_ value for an hedit control using a CONTROL statement:

```
CONTROL "ALC<402C>Dollars", IDD_PAID, "hedit", ES_LEFT | ... etc.
```

In the above example, the ALC<402C> value is stripped out with "Dollars" left as window text. The number 402C is the hexadecimal equivalent of:

```
ALC_NUMERIC | ALC_PUNC | ALC_MONETARY | ALC_GESTURE
```

The following example allows only kanji characters, katakana characters, and gestures; it does not specify initial window text:

```
CONTROL "ALC<74000>", IDD_J, "hedit", ES_LEFT | ... etc.
```

BXD_ Boxed Edit Control

The BXD_ values define the initial dimensions of the various components of a boxed edit (bedit) control. These are constants defined in terms of dialog units. They are converted to pixel dimensions by the bedit control before use.

For more information, see the entries for the **BOXLAYOUT** and **GUIDE** structures in Chapter 11, "Pen Application Programming Interface Structures."

The following table lists the BXD_ values.

Constant	Value	Description
BXD_BASEHEIGHT	13	Initial value for **cyBase** in **GUIDE** structure after conversion from dialog units to pixels.
BXD_BASEHORZ	0	Initial value for **cxBase** in **GUIDE** structure after conversion from dialog units to pixels.
BXD_CELLHEIGHT	16	Initial value for **cyBox** in **GUIDE** structure after conversion from dialog units to pixels.
BXD_CELLWIDTH	12	Initial value for **cxBox** in **GUIDE** structure after conversion from dialog units to pixels.
BXD_CUSPHEIGHT	2	Initial value for **cyCusp** in **BOXLAYOUT** structure after conversion from dialog units to pixels.
BXD_ENDCUSPHEIGHT	4	Initial value for **cyEndCusp** in **BOXLAYOUT** structure after conversion from dialog units to pixels.
BXD_MIDFROMBASE	0	Same as BXD_BASEHORZ.

BXDK_ Japanese Boxed Edit Control

The BXDK_ values define the initial dimensions of the various components of a Japanese boxed edit (bedit) control. These are constants defined in terms of dialog units. They are converted to pixel dimensions by the bedit control before use.

For more information, see the entries for the **BOXLAYOUT** and **GUIDE** structures in Chapter 11, "Pen Application Programming Interface Structures."

The following table lists the BXDK_ values.

Constant	Value	Description
BXDK_BASEHEIGHT	28	Initial value for **cyBase** in **GUIDE** structure after conversion from dialog units to pixels.
BXDK_BASEHORZ	0	Initial value for **cxBase** in **GUIDE** structure after conversion from dialog units to pixels.
BXDK_CELLHEIGHT	32	Initial value for **cyBox** in **GUIDE** structure after conversion from dialog units to pixels.
BXDK_CELLWIDTH	32	Initial value for **cxBox** in **GUIDE** structure after conversion from dialog units to pixels.
BXDK_CUSPHEIGHT	28	Initial value for **cyCusp** in **BOXLAYOUT** structure after conversion from dialog units to pixels.
BXDK_ENDCUSPHEIGHT	10	Initial value for **cyEndCusp** in **BOXLAYOUT** structure after conversion from dialog units to pixels.

IDC_ Display Cursor

A pen-aware display driver must define the following new cursor types.

Constant	Value	Description
IDC_ALTSELECT	32501	Upside-down standard arrow used for tap-and-hold selection.
IDC_PEN	32631	Default pen. Pen points up and to the left.

Example

You can access the tap-and-hold cursor with the following code:

```
HANDLE     hPenDLL = GetSystemMetrics( SM_PENWINDOW);
if (hPenDLL)
     SetCursor( LoadCursor( hPenDLL, IDC_ALTSELECT ) );
```

PCM_ Pen Collection Mode

Pen collection mode values define the condition that terminates an input session. (The system is said to be in "pen collection mode" during an input session when pen movement generates input data instead of being interpreted as mouse movement.) Pen collection can be stopped on any of the following conditions set by the PCM_ values in the **dwPcm** member of the **PCMINFO** structure:

Constant	Description
PCM_ADDDEFAULTS	Combine the default termination conditions with those specified by the application.
PCM_DOPOLLING	Request polling mode, rather than the default WM_PENEVENT messages.
PCM_INVERT	Stop pen collection if the user touches the "eraser" end of the pen to the tablet. Not all tablets can detect this event.
PCM_PENUP	Stop pen collection when the pen is lifted from the tablet.
PCM_RANGE	Stop pen collection when the pen leaves tablet's range of sensitivity. Not all tablets can detect this event.
PCM_RECTBOUND	Stop when the pen is placed down outside the inclusion rectangle. The inclusion rectangle is specified in the **rectBound** member of the **PCMINFO** structure.
PCM_RECTEXCLUDE	Stop when the pen touches inside the exclusion rectangle. The exclusion rectangle is specified in the **rectExclude** member of the **PCMINFO** structure.
PCM_RGNBOUND	Stop when the pen touches outside the bounding region specified in the **hrgnBound** member of the **PCMINFO** structure.
PCM_RGNEXCLUDE	Stop when the pen touches inside the exclusion region specified in the **hrgnExclude** member of the **PCMINFO** structure.
PCM_TAPNHOLD	Enable detection of the tap-and-hold gesture.
PCM_TIMEOUT	Stop pen collection if there is no pen activity for a specified time-out. The time-out value is specified in the **dwTimeout** member of the **PCMINFO** structure.

PDC_ Pen Device Capabilities

The following table lists the values for the **lPdc** member of the **PENINFO** structure:

Constant	Description
PDC_BARREL1	Barrel button 1 is present.
PDC_BARREL2	Barrel button 2 is present.
PDC_BARREL3	Barrel button 3 is present.
PDC_INTEGRATED	Tablet surface is also a display monitor.
PDC_INVERT	The tablet can detect when the "eraser" end of the pen is in contact with the tablet.
PDC_PROXIMITY	The tablet can detect when the pen is near but not in contact with the tablet surface.
PDC_RANGE	The tablet can detect when the pen leaves or enters the tablet's range of sensitivity.
PDC_RELATIVE	The pen driver can generate only relative coordinates.

For additional details, see the entry for the **PENINFO** structure in Chapter 11, "Pen Application Programming Interface Structures."

PDK_ State Bits for Pen Driver Kit

The PDK_ values inform the system when a mouse event is being generated by pen movement, as well as the current state of any barrel buttons. This information is contained in the **wPDK** and **wPdk** members of the **STROKEINFO** and **PENPACKET** structures, respectively. The **GetPenAsyncState** function also accepts a PDK_ value as its only argument. The following table lists the PDK_ values:

Constant	Value	Description
PDK_NULL	0x0000	No flags set (default).
PDK_UP	0x0000	Same as PDK_NULL.
PDK_BARREL1	0x0002	Barrel button 1 is depressed.
PDK_BARREL2	0x0004	Barrel button 2 is depressed.
PDK_BARREL3	0x0008	Barrel button 3 is depressed.
PDK_DOWN	0x0001	Pen is in contact with the tablet.
PDK_SWITCHES	0x000F	All of the above.
PDK_TIPMASK	0x0001	Mask for testing PDK_DOWN.
PDK_TRANSITION	0x0010	Only has meaning if set by pen services. This bit is set if the first point in the sequence being returned is in a different pen-tip state (up or down) from the previous points returned. If set on a call to **AddPointsPenData**, a new stroke is created even if the previous call to **AddPointsPenData** appended points of the same pen state. By default, a sub-sequent call to **AddPointsPenData** adding points of the same state as the previous call appends the points to the last stroke instead of creating a new stroke.
PDK_EVENT	0x0010	Alias for PDK_TRANSITION.
PDK_PENIDMASK	0x0F00	Mask for bits 8-11 (see paragraph following table).
PDK_INVERTED	0x0080	Pen is upside down ("eraser" end is in contact with tablet).
PDK_INKSTOPPED	0x2000	Inking has stopped.
PDK_OUTOFRANGE	0x4000	Set if the tablet detects the pen leaving the range of detection. If set, other information in the packet is invalid.
PDK_DRIVER	0x8000	Set if event is generated by the pen driver (as opposed to the mouse driver).

For PDK_ values other than PDK_PENIDMASK, bits 8 through 11 contain the identification number of the physical pen that generated the event. Pen numbering begins at 0.

See Also

PENPACKET, **STROKEINFO**

PDT_ OEM-Specific Data

PDT_ values provide information specific to the tablet hardware. These values are used in the **wPdt** member of the **OEMPENINFO** structure.

Constant	Value	Description
PDT_NULL	0	Null value.
PDT_PRESSURE	1	Tablet can detect change in pen pressure.
PDT_HEIGHT	2	Tablet can detect height of pen above surface.
PDT_ANGLEXY	3	Tablet can detect change in pen horizontal angle.
PDT_ANGLEZ	4	Tablet can detect change in pen vertical angle.
PDT_BARRELROTATION	5	Tablet can detect rotation of pen barrel.
PDT_OEMSPECIFIC	16	Maximum number of values allowed.

For additional information, see the **PENINFO** and **OEMPENINFO** structures in Chapter 11, "Pen Application Programming Interface Structures."

PDTS_ Pen Data Scaling

The PDTS_ data scaling units are used in the *uScale* and *wPndtNew* arguments of the **CreatePenData** and **MetricScalePenData** functions, respectively. These units are used for the **wPndts** member of the **PENDATAHEADER** structure. Positive x-coordinate is to the right; positive y-coordinate is down.

The following table lists the PDTS_ values. These values cannot be combined with the bitwise-OR operator.

Constant	Description
PDTS_ARBITRARY	The application has done its own scaling of the data point.
PDTS_COMPRESS2NDDERIV	The second derivative between points is stored.
PDTS_DISPLAY	Each logical unit is equivalent to a display pixel. Positive x is to the right; positive y is down.
PDTS_HIENGLISH	Each logical unit is mapped to 0.001 inch. Positive x is to the right; positive y is down.
PDTS_HIMETRIC	Each logical unit is mapped to 0.001 millimeter. Positive x is to the right; positive y is down.
PDTS_LOMETRIC	Each logical unit is mapped to 0.01 millimeter. Positive x is to the right; positive y is down.
PDTS_STANDARDSCALE	The standard scaling metric, equivalent to PDTS_HIENGLISH. Standard recognizers scale to this value.

The following table lists the PDTS_ bit settings. These bit settings can be combined using the bitwise-OR operator.

Constant	Description
PDTS_COMPRESSED	The data is compressed.
PDTS_NOCOLLINEAR	All redundant points removed.
PDTS_NOEMPTYSTROKES	All empty strokes removed.
PDTS_NOOEMDATA	OEM data removed.
PDTS_NOPENINFO	The **PENINFO** structure has been trimmed from the header.
PDTS_NOTICK	Timing information removed.
PDTS_NOUPPOINTS	All pen-up strokes removed.
PDTS_NOUSER	User information removed.

The following table lists the PDTS_ mask values:

Constant	Description
PDTS_COMPRESSMETHOD	Bits encode the compression scheme that is used.
PDTS_SCALEMASK	Mask scaling bits for hardware information.

For additional information, see the entries for the **CompactPenData** and **MetricScalePenData** functions in Chapter 10, "Pen Application Programming Interface Functions."

PDTT_ Pen Data Trimming

PDTT_ values are used as arguments for the **CompactPenData** function. The following table describes the PDTT_ values:

Constant	Description
PDTT_DEFAULT	Reallocates memory block to fit data; does not trim the data.
PDTT_ALL	Removes **PENINFO** structure from header, all pen-up points, OEM data, and collinear points.
PDTT_COLLINEAR	Removes coincident and collinear points from the pen data.
PDTT_COMPRESS	Compresses the data without losing any information.
PDTT_DECOMPRESS	Decompresses the data.
PDTT_OEMDATA	Removes all OEM data.
PDTT_PENINFO	Removes **PENINFO** structure from header.
PDTT_UPPOINTS	Removes all data from pen-up points (points collected when the pen is not in contact with the tablet).

For additional information, see the **CompactPenData** function in Chapter 10, "Pen Application Programming Interface Functions."

PMI_ Pen Miscellaneous Information

The PMI_ values are used as arguments for the **GetPenMiscInfo** and **SetPenMiscInfo** functions. The WM_PENMISCINFO message also uses PMI_ values in its *wParam* parameter.

The following table describes the PMI_ values:

Constant	Description
PMI_BEDIT	Boxed edit information.
PMI_CXTABLET	Width of tablet (in units of 0.001 inch) if present, else the width of the screen.
PMI_CYTABLET	Height of tablet (in units of 0.001 inch) if present, else the height of the screen.
PMI_ENABLEFLAGS	Flags describing whether certain Pen API features are enabled. The flags can be a combination of the following values: PWE_AUTOWRITE Enable pen functionality where the I-Beam cursor is present. PWE_ACTIONHANDLES Enable action handles in controls. PWE_INPUTCURSOR Show cursor while writing. PWE_LENS Enable pop-up letter guides (that is, the lens).
PMI_INDEXFROMRGB	A standard RGB pen-tip color value from 0 to 0xFFFFFF.
PMI_PENTIP	Address of current pen-tip structure.
PMI_RGBFROMINDEX	An integer index from 0 to 15 for the standard pen-tip color table.
PMI_SAVE	Save settings to file.
PMI_SYSFLAGS	Flags describing which pen system components are loaded. The flags can be a combination of the following values: PWF_RC1 Support available for Pen API version 1.0 Recognition Context (RC) and associated functions. PWF_PEN Pen/tablet hardware is present. PWF_INKDISPLAY Ink-compatible display driver is present. PWF_RECOGNIZER System recognizer is present. PWF_BEDIT Boxed edit (bedit) control is available. PWF_HEDIT Handwriting edit (hedit) control is available. PWF_IEDIT Ink edit (iedit) control is available. PWF_ENHANCED Enhanced features, including gesture support and 1 millisecond timing, are available. PWF_FULL All components listed above

	are present.
PMI_SYSREC	Handle to system recognizer, if present.
PMI_TICKREF	Absolute reference time that the system uses to calculate time-stamps for strokes in pen data objects and inksets.
PMI_TIMEOUT	Time-out value to end handwriting input, in milliseconds.
PMI_TIMEOUTGEST	Time-out value to end a gesture, in milliseconds.
PMI_TIMEOUTSEL	Time-out value in milliseconds for press-and-hold gesture. The range of permissible values is 0 to 5000. If press-and-hold has been disabled, this value is 65,535.

For additional information, see the entries for the **GetPenMiscInfo** and **SetPenMiscInfo** functions in Chapter 10, "Pen Application Programming Interface Functions."

RCD_ Writing Direction

RCD_ values are used in the **wRcDirect** member of the global **RC** structure. The **RC** structure is passed to a version 1.0 recognizer in the *lpRC* argument of **InitRC** and informs the recognizer of the writing direction. To set the writing direction differently than the default direction, call **SetGlobalRC** with the desired RCD_ value in **wRcDirect**.

RCD_ constants are provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

The writing direction consists of both primary and secondary directions. For example, English is written from left to right (primary) and then down the page (secondary). Chinese is often written from the top down (primary) and then right to left across the page (secondary).

The high byte of the direction indicates primary direction; the low byte indicates secondary direction. A recognizer can choose to ignore this word and support only the natural direction of the given language. The default value is determined by the recognizer.

The following table lists the RCD_ values:

Constant	Description
RCD_DEFAULT	Default value.
RCD_BT	Bottom to top.
RCD_LR	Left to right.
RCD_RL	Right to left.
RCD_TB	Top to bottom.

Example

For example, the value for standard English writing direction is defined as follows:

```
#define wRcDirectRoman  ((RCD_LR<<8) | RCD_TB)
```

See Also

RC

RCO_ Recognition Options

RCO_ values apply only to recognizers compatible with version 1.0 of the Pen API. They are used in the **IRcOptions** member of the **RC** structure, which specifies various options for recognition. RCO_ values can be combined with a logical-OR operator.

Constant	Description
RCO_BOXCROSS	Display a plus sign (+) at center of each box in a bedit control.
RCO_BOXED	Set if the writer is expected to write in boxes and the **GUIDE** structure contains valid data.
RCO_COLDRECOG	Set in results messages if the result is coming from cold recognition.
RCO_DISABLEGESMAP	Disables gesture mapping during the **Recognize** function call.
RCO_NOFLASHCURSOR	No flash cursor feedback.
RCO_NOFLASHUNKNOWN	If set in the **RC** structure and nothing was recognized, the cursor will not momentarily change to a question-mark cursor shape.
RCO_NOHIDECURSOR	If set, doesn't remove cursor while inking.
RCO_NOHOOK	Prevents application-wide and system-wide hooks from being called.
RCO_NOPOINTEREVENT	If set, the RC Manager will not try to recognize a pointer event but will pass on all data to the recog-nizer. This is useful, for example, if the application has installed a shape recognizer so the user can enter dots of ink. If the null recognizer is selected into the **RC**, RCO_NOPOINTEREVENT is assumed to be set.
RCO_NOSPACEBREAK	If set, indicates that the results passed back from the recognizer should be passed on to the dictionaries without breaking at space boundaries.
RCO_SAVEALLDATA	Saves all the pen data in the **RCRESULT** structure that is generated by the tablet, including any data for pen-up strokes and optional data such as pressure. By default, only data used by the recognizer is saved. The Microsoft recognizer collects all data from first to last pen-down stroke, including pen-up strokes in between, and any available OEM data for each stroke.
RCO_SAVEHPENDATA	Saves the pen data. If this is set, the recognizer does not delete the data when the application returns from WM_RCRESULT. It is the application's responsibility to free the pen data.
RCO_SUGGEST	If set, the following actions take place:

	After all dictionaries have been unsuccessfully searched with strings from the symbol graph, each dictionary is called with DIRQ_SUGGEST to allow the dictionaries to make suggestions. If a string is not yet identified by a dictionary, the null dictionary is used to create a symbol string from the symbol graph.
RCO_TABLETCOORD	If set, indicates that the members representing coordi-nate values in the **RC** structure are in tablet coordi-nates instead of screen coordinates. This can be used to collect recognition data on the portion of the tablet not mapped to the screen.

RCOR_ Tablet Orientation

RCOR_ values are used in the **wRcOrient** member of the global **RC** structure. The **RC** structure is passed to a version 1.0 recognizer in the *lpRC* argument of **InitRC** and informs the recognizer of the tablet orientation. The recognizer can optionally use the orientation to direct the transformation of tablet coordinates to ideal coordi-nates used for recognition.

RCOR_ constants are provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

The following table lists the RCOR_ values:

Constant	X-coordinate	Y-coordinate
RCOR_NORMAL	X = X'	Y = Y'
RCOR_LEFT	X = yMax - Y'	Y = X'
RCOR_RIGHT	X = Y'	Y = xMax - X'
RCOR_UPSIDEDOWN	X = xMax - X'	Y = yMax - Y'

As with the preceding values, direction is provided as a clue to the recognizer. A recognizer may attempt to identify the direction of writing by itself.

See Also

RC

RCP_ User Preferences

RCP_ values are used in the **wRcPreferences** member of the global **RC** structure. The **RC** structure is passed to a version 1.0 recognizer in the *lpRC* argument of **InitRC** and informs the recognizer of the user preferences.

RCP_ constants are provided only for compatibility with version 1.0 of the Pen API and will not be supported in future versions.

The following table lists the RCP_ values:

Constant	Description
RCP_LEFTHAND	User writes with left hand.
RCP_MAPCHAR	Tells a version 1.0 recognizer to fill in segmentation information in the **lpsyc** member of the **SYG** structure. This value cannot be set by the user because there is no Control Panel access to it. RCP_MAPCHAR is used by the Trainer.

See Also

RC

RCRT_ Results Type

RCRT_ values apply only to recognizers compatible with version 1.0 of the Pen API. They are used in the **wResultsType** member of the **RCRESULT** structure, which specifies the type of results as described in the following table:

Constant	Description
RCRT_ALREADYPROCESSED	Set by a hook if the result has already been acted upon. If an application receives a result with this bit already set, it should erase the ink and perform no other processing. An application-wide hook can set this flag. The Hform sample application demonstrates its use.
RCRT_DEFAULT	Normal return type.
RCRT_GESTURE	Result is a gesture symbol.
RCRT_NORECOG	Nothing recognized; only the data is returned. No recognition was attempted.
RCRT_NOSYMBOLMATCH	Nothing recognized. The ink drawn did not match any enabled symbols.
RCRT_PRIVATE	Recognizer-specific symbol recognized.
RCRT_UNIDENTIFIED	Result contained unidentified results.

Example

The code below shows an example of how to use RCRT_ values:

```
if ((lpr->wResultsType & (RCRT_NOSYMBOLMATCH | RCRT_ALREADYPROCESSED |
                                        RCRT_NORECOG | RCRT_PRIVATE ) ) == 0 )
{
     // A gesture or character
     if (lpr->wResultsType & RCRT_GESTURE)
     {
                  .
                  .          // Handle Gesture
                  .
     }
     else
     {
                  .
                  .          // Character results
                  .
     }
}
else
{
     .          // Handle special cases as necessary.  In general,
     .          // should just ignore.  This is what hedits do.
```

.

}

REC_ Recognition Functions

The REC_ constants specify return values from the **GetPenHwEventData** and **GetPenHwData** functions. They are also returned from the obsolete functions **Recognize**, **RecognizeData**, and **ProcessWriting**, and as the *wParam* value of the WM_RCRESULT message. Return values less than REC_DEBUG are provided for debugging purposes only and represent abnormal termination.

Constant	Description
REC_OK	This result message to be followed by other results before **Recognize** terminates. This is a valid *wParam* value for WM_RCRESULT, but it can never be the return value for **Recognize**.
REC_ABORT	Recognition stopped by a call to **EndPenCollection** with this value. The *lpPnt* data is not valid.
REC_BADHPENDATA	Returned if **HPENDATA** cannot be locked or has an invalid header. This value is also returned if **HPENDATA** has no data in it or if the data is in an incorrect scale or compressed.
REC_BUFFERTOOSMALL	Returned by **GetPenHwEventData**.
REC_BUSY	Returned if another task is currently performing recognition.
REC_DONE	Returned by **RecognizeData** upon normal completion.
REC_NA	Function not available.
REC_NOINPUT	Returned by **RecognizeData** if the buffer contains no data, or returned by **Recognize** if recognition ended before any data was collected. For example, a pen-down stroke may have occurred outside the bounding rectangle before any data was collected.
REC_NOTABLET	Tablet not physically present.
REC_OOM	Out-of-memory error.
REC_OVERFLOW	Data overflow during execution of the call.
REC_POINTEREVENT	Returned if the user makes contact with the tablet surface and lifts the pen before the pen tip travels a short distance. This value is also returned if the user does a press-and-hold action; that is, the pen makes contact with the tablet and remains in that position for a short period of time. REC_POINTEREVENT informs the application it should begin selection actions rather than inking or recognition. If REC_POINTEREVENT is returned, no WM_RCRESULT message is generated and no ink is displayed.
REC_TERMBOUND	Recognition ended because of a hit test outside the bounding rectangle. The **pntEnd**

	member of **RCRESULT** is filled with the point causing the stop.
REC_TERMEX	Recognition ended because of a hit test inside the exclusion rectangle. The **pntEnd** member of **RCRESULT** is filled with the point causing the stop.
REC_TERMOEM	Values greater than or equal to 512 reserved for recognizer-specific termination reasons.
REC_TERMPENUP	Recognition ended on pen up. The**pntEnd** member of **RCRESULT** is filled with the pen-up point that terminated recognition.
REC_TERMRANGE	Recognition ended because the pen left the proximity range.
REC_TERMTIMEOUT	Recognition ended on time-out. (The pen was up con-tinuously for a given amount of time.)

Debugging Values

All of the values listed in the following table are in the debug version of PENWIN.DLL only. No WM_RCRESULT message is generated if these values are returned by **Recognize**.

Constant	Description
REC_DEBUG	All debugging return values are less than or equal to this.
REC_ALC	Invalid enabled alphabet.
REC_BADEVENTREF	Returned when the **wEventRef** member in the *RC* structure is invalid.
REC_CLVERIFY	Invalid verification level.
REC_DICT	Invalid dictionary parameters.
REC_ERRORLEVEL	Invalid error level.
REC_GUIDE	Invalid **GUIDE** structure.
REC_HREC	Invalid recognition handle.
REC_HWND	Invalid handle to window to send results to.
REC_INVALIDREF	Invalid data reference parameter.
REC_LANGUAGE	Returned by the recognizer when the **lpLanguage** member of **RC** contains a language that is not supported by the recognizer. Call **ConfigRecognizer** with the WCR_QUERYLANGUAGE subfunction to determine whether or not a particular language is supported.
REC_NOCOLLECTION	Returned by **GetPenHwData** if collection mode has not been set.
REC_OEM	Error values below REC_OEM (-1024) are specific to the recognizer.
REC_PCM	Invalid **lPcm** member in **RC** structure. There is no way for the recognition to end.
REC_PARAMERR	Invalid parameter.
REC_RECTBOUND	Invalid rectangle.
REC_RECTEXCLUDE	Invalid rectangle.
REC_RESULTMODE	Unsupported results mode requested.

SYV_ Symbol Values

Each glyph a recognizer can identify has an associated symbol value. It is this value that is returned to the application by the recognizer.

The high-order and low-order words of a symbol value have the following meanings:

High-order word	Low-order word
0	System symbols.
1	ANSI character code.
2	Gestures.
3	Shift character codes (kanji).
4	Shapes.
5	Unicode.
6	Virtual keys.
7-0x7EFF	Reserved for future use.
0x7F00-0x7FFF	Recognizer-specific symbols.
>=0x8000	Character code for given code page. The low 15 bits of the high-order word indicate the code page.

Recognizers for the European market should return symbol values using ANSI and gesture symbol values (ANSI is the native character set for Windows in the European market). For the Japanese market, recognizers can use Shift JIS Level 1 and gestures. When writing a recognizer, bear in mind that symbol values outside these ranges cannot be interpreted by all Windows applications.

System Symbol Values

The following system symbol values are supported for recognizers:

Constant	Description
SYV_BEGINOR	Begins a list of choices; in the bedit guide, displayed as an opening brace character ({).
SYV_EMPTY	Empty.
SYV_ENDOR	Ends a list of choices. In the bedit guide, displayed as a closing brace character (}).
SYV_NULL	Null terminator.
SYV_OR	Separator for list of choices; in the bedit guide, displayed as a vertical bar (|).
SYV_SOFTNEWLINE	Translated to a space by **SymbolToCharacter**.
SYV_SPACENULL	Used in a symbol graph to indicate an alternative to a space.
SYV_UNKNOWN	Unrecognized glyph.

Gesture Symbol Values

All system recognizers are expected to recognize a special set of glyphs used as commands. In the following table, the "Windows Equivalent" column shows the mouse and keyboard equivalents in Windows.

Constant	Value	Description	Windows equivalent
. . .	00-0x00FF	Command gesture given. The low byte specifies which ANSI character was modified by the command gesture.	Nonstandard (usually CTRL+key).
SYV_BACKSPACE	0x00020008	Deletes character under gesture and sets insertion point.	BACKSPACE.
SYV_CLEAR	0x0002FFD5	Clears the selection.	DEL.
SYV_CLEARCHAR	0x0002FFD2	Clears the selection.	DEL.
SYV_CLEARWORD	0x0002FFDD	Deletes word or object under gesture.	Double-click, DEL.
SYV_COPY	0x0002FFDA	Copies selection to Clipboard.	CTRL+INS.
SYV_CORRECT	0x0002FFDF	Corrects selection or word under gesture.	None.
SYV_CONTEXT	0x0002FFD7	Displays a context menu	Right mouse click.
SYV_CUT	0x0002FFDB	Cuts selection and places it on Clipboard.	SHIFT+DEL.
SYV_EXTENDSELECT	0x0002FFD8	For linear selection (text), selects all text between current insertion point and point of extend-selection gesture. For nonlinear selection (objects), adds object under gesture to selection.	SHIFT+mouse click.
SYV_INSERT	0x0002FFD6	Opens the lens to allow text to be inserted.	None
SYV_KKCONVERT	0x0002FFD4	Starts Kana-to-Kanji converter. (Japanese version only.)	N/A
SYV_PASTE	0x0002FFDC	Pastes selection at point indicated by hot spot of paste gesture.	Click (place insertion point) followed by SHIFT+INS.
SYV_RETURN	0x0002000D	Enters carriage-return key.	Click, RETURN.
SYV_SPACE	0x00020020	Adds space character.	Click, SPACEBAR.
SYV_TAB	0x00020009	Enters tab.	Click, TAB.
SYV_UNDO	0x0002FFD9	Undoes previous action.	ALT+BACKSPACE.
SYV_USER	0x0002FFDE	Any circle gesture.	See the following section.

Circle Gesture Symbol Values

A circle gesture consists of a circled letter, either uppercase or lowercase. The Pen API version 2.0 does not explicitly support circle gestures. They are defined only for application or recognizer use.

The following table lists the SYV_ values for the circle gestures. Intervening values correspond to the letters between "a" and "z".

Constant	Value	Description
SYV_APPGESTUREMASK	0x00020000	Mask value for circle gestures.
SYV_CIRCLELOA	0x000224D0	Lowercase "a" circle gesture.
SYV_CIRCLELOZ	0x000224E9	Lowercase "z" circle gesture.
SYV_CIRCLEUPA	0x000224B6	Uppercase "A" circle gesture.
SYV_CIRCLEUPZ	0x000224CF	Uppercase "Z" circle gesture.

Selection Symbol Values

The following table lists the SYV_values for selection symbol gestures.

Constant	Value	Description
SYV_SELECTFIRST	0x0002FFC2	Minimum value for section SYVs.
SYV_LASSO	0x0002FFC1	Lasso selection (equivalent to double-click).
SYV_SELECTLEFT	0x0002FFC2	Select text to the left (not supported by the Microsoft Handwriting Recognizer).
SYV_SELECTRIGHT	0x0002FFC3	Select text to the right (not supported by the Microsoft Handwriting Recognizer).
SYV_SELECTLAST	0X0002FFCF	Maximum value for selection SYVs.

Shape Symbol Values

These values should be used by shape recognizers.

Constant	Value	Description
SYV_SHAPELINE	0x00040001	Shape recognized as a line.
SYV_SHAPEELLIPSE	0x00040002	Shape recognized as an ellipse.
SYV_SHAPERECT	0x00040003	Shape recognized as a rectangle.
SYV_SHAPEMIN	0x00040001	Minimum value for recognized shape SYVs.
SYV_SHAPEMAX	0x00040003	Maximum value for recognized shape SYVs.

Appendix Differences Between Versions 1.0 and 2.0 of the Pen Application Programming Interface

Version 2.0 of the Pen Application Programming Interface (API) provides more services – and more avenues for innovation – than did version 1.0. A skimming of Chapters 10 and 11, which identify functions and structures new to version 2.0, indicates the extent of the enhancements added to the API.

However, if you have used version 1.0 of the Pen API, also known as Microsoft Windows for Pen Computing, you will find more than additional functions and other services in this release. Programming philosophy has changed as well. Partic-ularly in the area of recognition, the Pen API now allows greater freedom and responsibility for the handling and interpretation of pen input.

This appendix identifies some of the most important changes and improvements to the Pen API since version 1.0. It would require a number of pages to itemize all the improvements incorporated into version 2.0, which are described throughout this book. The purpose of this appendix is to help the developer familiar with version 1.0 to quickly identify several important areas of version 2.0 that reflect significant change. You will find that version 2.0 of the Pen API opens up new possibilities for collected ink data other than simply passing it to a recognizer.

Improvements to the bedit Control

The bedit control of version 2.0 of the Pen API has been significantly improved over that of version 1.0. The improvements aim to make text entry more convenient and more intuitive for the user. The following list briefly describes the major improvements:

- The current insertion point is now indicated by an action handle instead of the caret of version 1.0.
- The user can select text by dragging the insertion point action handle. Selected text appears in reverse video.
- The user can move selected text within the control window by dragging it to a new position.
- Single-line bedit windows can automatically scroll horizontally when the user fills either of the last two visible boxes. The last character remains visible after the scroll to help orient the user. The control window also provides scroll arrows for horizontal scrolling.
- A single tap near the center of a character displays a list of alternative characters determined by the recognizer. Double-tapping brings up a menu with a list of alternative words that can replace the entire word above the tap. The user selects a letter or complete word by tapping the menu selection.
- As do other controls in Microsoft Windows 95, bedit controls now provide a context menu from which the user can cut or copy a selection, paste, insert, and so forth.
- If the user inserts a carriage return in a line of a multiline bedit, text to the right of the carriage return automatically wraps to the next line.
- Empty cells are marked with a light-gray dot to help distinguish them from spaces.

The EM_LIMITTEXT message has a slightly different effect on bedit controls in version 2.0 of the Pen API. In version 1.0, sending EM_LIMITTEXT to a bedit control window set the number of boxes in the control as specified by the message's *wParam* parameter. In version 2.0, EM_LIMITTEXT sets the maximum number of bytes of text the control can hold instead of the number of boxes. For more information about EM_LIMITTEXT, see "The bedit Control" in Chapter 3, "The Writing Process."

Recognition

Version 2.0 of the Pen API significantly changes the way an application interacts with a recognizer. It allows an application to install multiple recognizers and use them selectively, first creating an **HRC** object for each recognizer to configure the recognition process. Version 2.0 provides many more recognition functions than did version 1.0, but places the full burden of recognition on the recognizer. All recognition functions are provided by the installed recognizer dynamic-link libraries (DLLs) and not by Windows.

The change in philosophy mentioned at the beginning of this chapter is particulary true with regard to recognition. Recognizers are now more clearly separate from the system and enjoy a corresponding freedom in their implementation. The Pen API defines the interaction between application and recognizer, but stops short of man-dating how a recognizer performs its tasks. The recognizer objects described in Chapter 8, "Writing a Recognizer," are called objects to emphasize that their forms are invisible to the application. The objects are "black box" entities, which the recognizer developer designs without restraint from the system.

For a description of the **HRC** object and other elements of recognition, see Chapter 5, "The Recognition Process."

The RC Structure

The core of the recognition process in version 1.0 was the **RC** data structure. The structure still exists in version 2.0, but it is made obsolete by the **HRC** object that governs the recognition process. An application can still use an **RC** structure when calling the obsolete version 1.0 functions **ProcessWriting**, **InitRC**, **Recognize**, and **RecognizeData**.

The following table lists the members of the **RC** structure. For each member, the second column describes the corresponding services in version 2.0 of the Pen API. Use this table to update your version 1.0 applications to use the new services. See the reference chapters in Part 2 for descriptions of the functions, messages, and constants cited in the table.

RC member	**Equivalent service in Pen API 2.0**
hrec	Return value from **InstallRecognizer**.
hwnd	PE_SETTARGETS message. (See "Step 2: PE_SETTARGETS Message" in Chapter 2.)
wEventRef	Return value from **GetMessageExtraInfo**. (See sample code in "InputWndProc" in Chapter 7.)
wRcPreferences	**ConfigHREC** with WCR_GETHAND or WCR_SETHAND.
lRcOptions	Various **HRC** functions described in Chapters 5 and 8.
lpfnYield	Not applicable in version 2.0.
lpUser	**GetPenMiscInfo** with PMI_USER. Cannot set new user in version 2.0.
wCountry	**GetInternationalHRC** or **SetInternationalHRC**.
wIntlPreferences	**ConfigHREC** with WCR_GETANSISTATE or WCR_SETANSISTATE.
lpLanguage	**GetInternationalHRC** or **SetInternationalHRC**.
rglpdf	Not applicable in version 2.0.
wTryDictionary	Not applicable in version 2.0.
clErrorLevel	Not applicable in version 2.0.
alc	**GetAlphabetHRC** or **SetAlphabetHRC**. (See "Specifying an Alphabet Set" in Chapter 8.)
alcPriority	**GetAlphabetPriorityHRC** or **SetAlphabetPriorityHRC**. Also **ConfigHREC** with WCR_GETALCPRIORITY or WCR_SETALCPRIORITY.
rgbfAlc	**GetAlphabetHRC** or **SetAlphabetHRC**.
wResultMode	Not applicable in version 2.0.
wTimeOut	**GetPenMiscInfo** or **SetPenMiscInfo** with PMI_TIMEOUT.
lPcm **rectBound** **rectExclude**	These three members are replaced by PE_GETPCMINFO message. (See "Step 3: PE_GETPCMINFO Message" in Chapter 2.)
guide	**GetGuideHRC** or **SetGuideHRC**.
wRcOrient	Not applicable in version 2.0.

wRcDirect	**ConfigHREC** with WCR_GETDIRECTION or WCR_SETDIRECTION.
nInkWidth **rgbInk**	These two members are replaced by **GetPenMiscInfo** and **SetPenMiscInfo** with PMI_PENTIP.
dwAppParam	Determined by recognizer.
dwDictParam	Not applicable in version 2.0.
dwRecognizer	Determined by recognizer.

The RCRESULT Structure

The **RCRESULT** structure provides the means for a version 1.0 recognizer to communicate results to the application. When the recognizer finishes its work, the application receives a WM_RCRESULT message containing a pointer to an **RCRESULT** structure. It then reads the recognizer's guesses from the structure.

In version 2.0, the **RCRESULT** structure is made obsolete by the **HRCRESULT** object. Although they have similar names, do not confuse the two or attempt to draw parallels between them. The **HRCRESULT** contains the recognizer's results in a format determined by the recognizer. This format most likely has nothing to do with **RCRESULT**. The application simply calls into the recognizer for recognition results.

For complete descriptions of the version 2.0 recognition functions, see Chapters 5 and 8 and the reference entries in Chapter 10.

Default Recognition

The version 2.0 application collects, displays, and distributes ink to the recognizer while the user is writing. The Pen API provides convenient and flexible default processing for these tasks in the **DoDefaultPenInput** function, which completely supersedes **ProcessWriting**.

Nearly all pen-based applications should take advantage of the capabilities of **DoDefaultPenInput**. Through a system of messages, the function allows an application to monitor and govern the recognition process, or simply accept the default decisions of **DefWindowProc**.

For a description of **DoDefaultPenInput** and the message traffic it generates, read Chapter 2, "Starting Out with System Defaults." To see **DoDefaultPenInput** in use, refer to the code for the PENAPP sample application presented in Chapter 7, "A Sample Pen Application."

Recognition Processing

In version 2.0, applications have much greater control over scheduling the recog-nition process. If it does not use **DoDefaultPenInput**, an application continuously feeds pen data to a recognizer through the **AddPenInputHRC** function. By calling **ProcessHRC**, the application can also schedule regular time slots for the recog-nizer to see the input as the user writes.

This real-time recognition processing contrasts with the recognition procedures of version 1.0, in which the application relinquished control to the obsolete **Recognize** function for the duration of the input session. The version 2.0 recognizer can now "cook" pen input virtually as it arrives from the pen driver, with the application determining how often and how long the recognizer has control.

Initializing and Closing a Recognizer

The recognition functions **InitRecognizer** and **CloseRecognizer** are obsolete in version 2.0 of the Pen API. In their place, two new subfunctions have been added to **ConfigRecognizer**. When the pen system loads a recognizer, it now calls **ConfigRecognizer** with the subfunction WCR_INITRECOGNIZER. In response to this call, the recognizer should perform the required initialization tasks formerly conducted by **InitRecognizer**.

Similarly, the system also calls **ConfigRecognizer** when it unloads the recognizer, this time with the subfunction WCR_CLOSERECOGNIZER. This call informs the recognizer it is being unloaded and it should conduct any required cleanup operations.

Applications based on version 2.0 do not call **ConfigRecognizer**. This is because the function has no argument that refers to a specific recognizer of the several that may be currently installed. Instead, version 2.0 applications call the new API function **ConfigHREC** to configure a recognizer or query for its capabilities. The system determines the intended recognizer and passes the call on to that recog-nizer's **ConfigRecognizer** function. Thus, a version 2.0 recognizer exports **ConfigRecognizer**, which an application accesses by calling **ConfigHREC**.

Word Lists and Dictionaries

Word lists are new to version 2.0. An application can select from among any number of word lists to help a recognizer verify its guesses. Word-list files must have standard text formatting to allow users to create or modify them with a text editor, but otherwise have no restrictions in size or content.

Dictionaries existed in version 1.0 as DLL files. In version 2.0, a dictionary serves a recognizer invisibly as a large word list. The application has no access to a dic-tionary except to tell the recognizer whether or not to use one. Dictionary files can have any format, but are usually compressed in some manner private to the recog-nizer.

A dictionary is thus a private (and optional) tool of a recognizer. The "system" in the name **EnableSystemDictionaryHRC** does not refer to the operating system, but simply emphasizes ownership. In this case, "system" means "not application."

Gestures

Version 2.0 no longer provides explicit support for user-defined gestures. The burden of recognizing and handling new gestures instead falls to the recognizer and application. The Gesture Manager (GESTMAN.EXE) and API function **ExecuteGesture** do not exist in version 2.0.

The standard set of "circle-letter" gestures remains, however. All version 2.0 edit controls – hedit, bedit, and iedit – respond normally to the gestures listed in the following table. An application or recognizer is free to provide support for any other gesture.

This table lists the gestures available in version 2.0 edit controls. Note that the V-circle and check-mark-circle gestures have identical behavior. The V-circle gesture is provided only to prevent confusion with the check-mark-circle gesture.

Gesture	Name	Action
{ewc ï¿½}	Lasso-tap	Select
{ewc ï¿½}	X-circle	Cut current selection
{ewc ï¿½}	C-circle	Copy current selection
{ewc ï¿½}	P-circle	Paste
{ewc ï¿½}	Check-mark-circle	Edit/Properties
{ewc ï¿½}	V-circle	Edit/Properties
{ewc ï¿½}	Caret-circle	Insert text
{ewc ï¿½}	M-circle	Display context menu
{ewc ï¿½}	D-circle	Clear/Delete
{ewc ï¿½}	S-circle	Space
{ewc ï¿½}	N-circle	Newline (carriage return)
{ewc ï¿½}	T-circle	Tab
{ewc ï¿½}	U-circle	Undo

Action Handles

In version 2.0, edit controls have small icons called *action handles*. Action handles provide the user a more intuitive and discoverable means for carrying out certain editing tasks than do gestures. With action handles, the user can:

- Select text.
- Cut, copy, and paste.
- Change the insertion point.
- Drag-and-drop a selection.
- Access a menu of options.

On-Screen Keyboard

In version 2.0, the on-screen keyboard is an independent application named SK.EXE, which can be launched in the same way any application is launched in Windows; for example, by using the **WinExec()** function.

The **ShowKeyboard** function, which invoked the on-screen keyboard in version 1.0, is still supported but for older applications, but has been modified due to the fact that the on-screen keyboard is a separate application in version 2.0.

The WM_SKB message, available in version 1.0, is no longer sent to top-level windows when the on-screen keyboard changes.

Timing Information

The new **HINKSET** object allows an application to refer to stroke data completely or partially by time, rather than by coordinates. The **HINKSET** object is a tem-poral version of **HPENDATA**. It sees each stroke as an interval of time instead of a collection of physical points. See "The HINKSET Object" in Chapter 4 for a description of stroke timing and **HINKSET**.

Stroke timing allows a new characteristic of ink rendering called *animation*. Through animation, an application can control the speed at which pen data is displayed. For more information, see the description of **DrawPenDataEx** in Chapter 10, "Pen Application Programming Interface Functions."

Targeting

An application based on version 2.0 of the Pen API can create multiple windows and writing areas on the screen that simultaneously accept pen input. Targeting allows the application to govern the distribution of input data to the proper window.

For example, a forms application can establish targeting information for each of several controls on the screen. Even if the user writes in the controls in arbitrary order, targeting ensures the pen data arrives at the proper window procedure.

For more information on targeting, see "Step 2: PE_SETTARGETS Message" in Chapter 2, "Starting Out with System Defaults." Also see the reference sections for **TARGET** and **TARGINFO** in Chapter 11, "Pen Application Programming Interface Structures."

HPENDATA Memory Block

The internal structure of the **HPENDATA** memory block has changed since ver-sion 1.0. As described in "Data Within an HPENDATA Object" in Chapter 4, the stroke header no longer incorporates a **STROKEINFO** structure. However, to maintain compatibility with version 1.0, the **GetPenDataStroke** function provides a copy of a **STROKEINFO** structure for the requested data points.

Since the **HPENDATA** format may again change in future versions, applications should avoid attempting to read the memory block directly and instead rely on the appropriate **HPENDATA** functions described in Chapter 4, "The Inking Process."

The **PENINFO** structure in the block's **HPENDATAHEADER** has also changed since version 1.0. **PENINFO** contains a new member, **fuOEM**, that indicates the type of original equipment manufacturer (OEM) data the **HPENDATA** object contains.

Registry Configuration

In version 1.0, the PENWIN.INI file contained system configuration information such as the name of the default recognizer and the time-out value for selection. Version 2.0 removes configuration information from the PENWIN.INI file and instead stores it in the system registry.

For example, the following code fragment sets the fictitious recognizer RECOG1.DLL as the default system recognizer. The code presumes that RECOG1.DLL:

- Resides in a location where Windows can find it (usually the system subdirectory).
- Is capable of taking on the role of system recognizer.

For more information about recognizers, see Chapter 8, "Writing a Recognizer." For descriptions of the registry functions **RegSetValueEx**, **RegCreateKey**, and **RegCloseKey**, see the documentation included in the Win32 Software Develop-ment Kit.

```
HKEY    hk;             // Key handle returned by RegCreateKey
    .
    .
    .
// Open (or create) the registry for the parent key
if ((lRes = RegCreateKey( HKEY_LOCAL_MACHINE,
                          (LPSTR)REGSTR_PATH_CONTROL,
                          &hk )) == ERROR_SUCCESS)
{
    // If successful, set "RECOG1.DLL" as the system recognizer
    RegSetValueEx( hk, REGSTR_VAL_PEN_RECOG, NULL, REG_SZ,
                (LPBYTE)(LPSTR)"RECOG1.DLL", 0 );

    // Close it
    RegCloseKey( hk );
}
```

The PENREG.H header file defines values pertaining to the system registry. Note that, generally, applications should not change the system configuration, relying instead on the user to do so through the Control Panel.

For information on retrieving and setting other pen system parameters, see the reference entries for **GetPenMiscInfo** and **SetPenMiscInfo** in Chapter 10, "Pen Application Programming Interface Functions."

Appendix Using the 32-Bit Pen Application Programming Interface

This appendix describes the 32-bit pen services provided by the PENWIN32.DLL and PKPD32.DLL libraries. With some exceptions, these dynamic-link libraries (DLLs) offer Win32™-based applications the same pen-based support as the 16-bit libraries PENWIN.DLL and PKPD.DLL without requiring the intermediate steps of thunk conversions.

The services not supported by the 32-bit Pen Application Programming Interface (API) consist mainly of outdated functions no longer required in version 2.0. These outdated functions are supported in the 16-bit Pen API only to maintain backward compatibility with version 1.0. They will not be supported in future versions.

To enable 32-bit pen services, an application must be created in a 32-bit environ-ment – that is, the compiler, linker, libraries, and header files must be of 32-bit type. Before including pertinent header files, the application must define the constant WIN32 as shown here:

```
#define    WIN32
#include   <windows.h>
#include   <penwin.h>
```

A 32-bit application links to the pen system libraries in the same way it links to any other Windows library, with either of the following methods:

- Link to the import library PKPD32.LIB (not recommended for PENWIN32.LIB) , or
- Explicitly load PENWIN32.DLL and PKPD32.DLL with the **LoadLibrary** function. After loading the libraries, the application must call **GetProcAddress** to obtain the address of each Pen API function it intends to use. Before termin-ating, the application should call **FreeLibrary** to unload the libraries from memory.

The first method should normally be used to link functions in PKPD32.LIB. This method can be used for functions in PENWIN32.LIB if it is known for certain that the application will always be run on systems that have pen services installed, in which case the application should test for the existence of pen services at startup and exit if not found.

The second method should be used for PENWIN32.DLL functions when it is anticipated that the application may be run on systems where pen services are not installed. The reason for not linking PENWIN32.LIB is to prevent the application from loading PENWIN32.DLL on a system that has not loaded pen services at startup. This could happen, for example, on a computer that has PENWIN32.DLL on the path but has not installed pen services. Loading PENWIN32.DLL dynamic-ally does not sufficiently start pen services and errors are likely to occur.

In a 32-bit application, the call

```
GetSystemMetrics( SM_PENWINDOWS );
```

returns the handle of PENWIN32.DLL. This DLL makes available some of the same resources (such as cursors) provided by the 16-bit PENWIN.DLL library.

32-Bit Functions

The following table lists the functions supported by the 32-bit Pen API. The syntax for each function remains the same as for 16-bit applications. For the description and syntax of each function, refer to Chapter 10, "Pen Application Programming Interface Functions."

The table also uses an asterisk (*) to identify the functions PKPD32.DLL exports. These functions are always available to 32-bit applications running with Windows 95, regardless of whether the PENWIN32.DLL file is present. For more informa-tion, see "Pen Kernel Functions" in Chapter 9, "Summary of the Pen Application Programming Interface."

AddInksetInterval*

AddPenDataHRC

AddPenInputHRC

AddPointsPenData*

AddWordsHWL

BoundingRectFromPoints*

CharacterToSymbol

CompressPenData*

ConfigHREC

ConfigRecognizer

CorrectWriting

CorrectWritingEx

CreateCompatibleHRC

CreateHWL

CreateInkset*

CreateInksetHRCRESULT

CreatePenData

CreatePenDataEx*

CreatePenDataHRC

CreatePenDataRegion*

DestroyHRC

DestroyHRCRESULT

DestroyHWL

DestroyInkset*

DestroyPenData*

DoDefaultPenInput

DPtoTP

DrawPenDataEx*

DuplicatePenData*

EnableGestureSetHRC

EnableSystemDictionaryHRC

EndPenInputHRC

ExtractPenDataPoints*

ExtractPenDataStrokes*

GetAlphabetHRC

GetAlphabetPriorityHRC

GetAlternateWordsHRCRESULT

GetBoxMappingHRCRESULT

GetBoxResultsHRC

GetGuideHRC

GetHotspotsHRCRESULT

GetHRECFromHRC

GetInksetInterval*

GetInksetIntervalCount*

GetInternationalHRC

GetMaxResultsHRC

GetPenAppFlags

GetPenAsyncState

GetPenDataAttributes*

GetPenDataInfo*

GetPenInput

GetPenMiscInfo

GetPenResource

GetPointsFromPenData*

GetResultsHRC

GetStrokeAttributes*

GetStrokeTableAttributes*

GetSymbolCountHRCRESULT

GetSymbolsHRCRESULT

GetVersionPenWin

GetWordlistCoercionHRC

GetWordlistHRC

HitTestPenData*

InsertPenData*

InsertPenDataPoints*

InsertPenDataStroke*

InstallRecognizer

IsPenEvent

MetricScalePenData*

OffsetPenData*

PeekPenInput

PenDataFromBuffer*

PenDataToBuffer*

ProcessHRC

ReadHWL

RedisplayPenData*

RemovePenDataStrokes*

ResizePenData*

SetAlphabetHRC

SetAlphabetPriorityHRC

SetBoxAlphabetHRC

SetGuideHRC

SetInternationalHRC

SetMaxResultsHRC

SetPenAppFlags*

SetPenMiscInfo

SetResultsHookHREC

SetStrokeAttributes*

SetStrokeTableAttributes*

SetWordlistCoercionHRC

SetWordlistHRC

StartInking

StartPenInput

StopInking

StopPenInput

SymbolToCharacter

TargetPoints

TPtoDP

TrainHREC

TrimPenData*

UnhookResultsHookHREC

UninstallRecognizer

WriteHWL

32-Bit Messages

The 32-bit Pen API does not support all the 16-bit messages described in Chap-ter 12, "Pen Application Programming Interface Messages." This section lists the messages that are supported by the 32-bit Pen API. Although most 32-bit messages behave the same way as 16-bit messages, certain WM_PENEVENT submessages behave differently.

WM_PENEVENT Submessages

The following submessages of WM_PENEVENT require different treatment in 32-bit applications:

PE_BEGININPUT

PE_BUFFERWARNING

PE_PENDOWN

PE_PENMOVE

PE_PENUP

PE_TERMINATED

PE_TERMINATING

In 16-bit applications, these submessages store different values in the high-order and low-order words of their *lParam*:

- *wParam* = PE_ submessage number
- LOWORD(*lParam*) = event reference or termination code
- HIWORD(*lParam*) = **HPCM** handle

This scheme is not possible in Win32-based applications because *lParam* must contain a single 32-bit handle; therefore, the parameters for the submessages listed above are arranged differently in the 32-bit Pen API:

- LOWORD(*wParam*) = PE_ submessage number
- HIWORD(*wParam*) = event reference or termination code
- *lParam* = **HPCM** handle

To extract data from the parameters, use the following macros defined in the PENWIN.H header file. These macros render the differences in the parameters transparent to an application because they automatically adjust for 16-bit or 32-bit type of programs:

Macro	Description
HpcmFromWpLp	Retrieves **HPCM** handle returned from **StartPenInput**. If the application calls **DoDefaultPenInput**, that function calls **StartPenInput** internally.
EventRefFromWpLp	Retrieves event reference value for session returned from **GetMessageExtraInfo**. Both **DoDefaultPenInput** and **StartPenInput** take this value as their second argument.
TerminationFromWpLp	Retrieves a PCM_ value indicating the reason for termination from the PE_TERMINATED and PE_TERMINATING submessages.
SubPenMsgFromWpLp	Retrieves PE_ submessage value.

The macros take both *wParam* and *lParam* as arguments and automatically return the desired value for both 16-bit and 32-bit applications. For example:

```
HPCM      hpcm;           // HPCM handle created by StartPenInput
LONG      lInfo;          // Return value from GetMessageExtraInfo
int       iRet;           // Error code
     .
     .
     .
case WM_LBUTTONDOWN:
     lInfo = GetMessageExtraInfo( );
     if (IsPenEvent( msg, lInfo ))
     {
           iRet = DoDefaultPenInput( hwnd, LOWORD(lInfo) );
     }
           .
           .
           .
case WM_PENEVENT:
     switch( SubPenMsgFromWpLp( wParam ) );
     {
           case PE_PENDOWN:
                 hpcm  = HpcmFromWpLp( wParam, lParam );
                 //
                 // Note lInfo and EventRefFromWpLp( wParam, lParam )
                 // contain the same event reference value.
                 //
```

List of 32-Bit Window Messages

The following table lists the WM_ window messages and corresponding sub-messages supported by the 32-bit Pen API (submessages available in the Japanese version are indicated by an asterisk):

WM_ messages	Submessages
WM_PENMISCINFO	* HE_CANCELCONVERT
WM_PENCTL	HE_CHAROFFSET
	HE_CHARPOSITION
	HE_DEFAULTFONT
	HE_ENABLEALTLIST
	* HE_FIXKKCONVERT
	HE_GETBOXLAYOUT
	* HE_GETCONVERTRANGE
	HE_GETINFLATE
	HE_GETINKHANDLE*
	HE_GETKKCONVERT
	* HE_GETKKSTATUS
	HE_GETUNDERLINE
	HE_HIDEALTLIST
	* HE_KKCONVERT
	*
	HE_PUTCONVERTCHARHE_SETBO
	XLAYOUT
	* HE_SETCONVERTRANGE
	HE_SETINFLATE
	HE_SETINKMODE
	HE_SETUNDERLINE
	HE_SHOWALTLIST
	HE_STOPINKMODE
WM_ messages	**Submessages**
WM_PENMISC	PMSC_BEDITCHANGE
	PMSC_GETHRC
	PMSC_GETINKINGINFO
	PMSC_GETPCMINFOPMSC_GETSY
	MBOLCOUNT
	PMSC_GETSYMBOLS
	PMSC_INKSTOP
	* PMSC_KKCTLENABLE
	PMSC_LOADPW
	* PMSC_PENUICHANGE
	PMSC_SETHRC
	PMSC_SETINKINGINFO
	PMSC_SETPCMINFO
	PMSC_SETSYMBOLS
	* PMSC_SUBINPCHANGE
WM_CTLINIT	CTLINIT_BEDIT
	CTLINIT_HEDIT
	CTLINIT_IEDIT
WM_PENEVENT	PE_BEGINDATA
	PE_BEGININPUT
	PE_BUFFERWARNING
	PE_ENDDATA
	PE_ENDINPUT

PE_GETINKINGINFO
PE_GETPCMINFO
PE_MOREDATA
PE_PENDOWN
PE_PENMOVE
PE_PENUP
PE_RESULT
PE_SETTARGETS
PE_TERMINATED
PE_TERMINATING

List of 32-Bit iedit Control Messages

The following table lists the IE_ messages available for iedit controls in 32-bit applications:

IE_CANUNDO

IE_DOCOMMAND

IE_EMPTYUNDOBUFFER

IE_GETAPPDATA

IE_GETBKGND

IE_GETCOMMAND

IE_GETCOUNT

IE_GETDRAWOPTS

IE_GETERASERTIP

IE_GETFORMAT

IE_GETGESTURE

IE_GETGRIDORIGIN

IE_GETGRIDPEN

IE_GETGRIDSIZE

IE_GETINK

IE_GETINKINPUT

IE_GETINKRECT

IE_GETMENU

IE_GETMODE

IE_GETMODIFY

IE_GETNOTIFY

IE_GETPAINTDC

IE_GETPDEVENT

IE_GETPENTIP

IE_GETRECOG

IE_GETSECURITY

IE_GETSEL

IE_GETSELCOUNT

IE_GETSELITEMS

IE_GETSTYLE

IE_SETAPPDATA

IE_SETBKGND

IE_SETDRAWOPTS

IE_SETERASERTIP

IE_SETFORMAT

IE_SETGRIDORIGIN

IE_SETGRIDPEN

IE_SETGRIDSIZE

IE_SETINK

IE_SETINKINPUT

IE_SETMODE

IE_SETMODIFY

IE_SETNOTIFY

IE_SETPENTIP

IE_SETRECOG

IE_SETSECURITY

IE_SETSEL

IE_UNDO

Appendix Modifying the SYSTEM.INI File

This appendix describes the settings used in the SYSTEM.INI. The SYSTEM.INI file contains all of the editable information used by the Pen Application Program-ming Interface (API). All other system information concerning the Pen API version 2.0 is maintained in the Windows 95 system registry and will not require modifica-tion except by using the Control Panel Pen icon after Pen Services for Windows 95 is installed.

It is also unlikely that you will manually edit the pen entries in SYSTEM.INI in the Windows 95 environment. A setup information file (PENWIN.INF) provided by Microsoft contains the script that Windows 95 Setup uses to install Pen Services for Windows 95 as an optional system component. This script adds the entries shown in the table below to the SYSTEM.INI file. The one SYSTEM.INI entry that you may want to edit manually after Pen Services is installed is the level of non-critical errors reported. For this, you will have to manually edit the PenWinErrors entry in the [boot] section of SYSTEM.INI, using one of the values shown in the table below.

In versions of Windows previous to Windows 95, pen services were removed from a system by manually editing the SYSTEM.INI file and deleting the entries shown in the table below. This should not be done with Windows 95. Use the Control Panel Add/Remove Programs icon to remove Pen Services for Windows 95. This will automatically delete all the Pen Services entries from SYSTEM.INI.

The following table lists the pen-related entries for SYSTEM.INI

SYSTEM.INI Entry Made at Installation Time	Description
[boot]	Section name.
drivers=pen penwindows	Defines installable drivers.
PenWinErrors=1	Determines the level of non-critical errors displayed. 0 = show no errors or warnings. 1 = show errors only. 2 = show errors and warnings.
[drivers]	Section name.
pen=penc.drv	Sets the pen driver.
penwindows=penwin.dll	Pen API library.

Appendix Accessing the Pen Device Driver

There are no specific functions in the Pen Application Programming Interface (API) for pen driver use. Instead, the pen driver functionality is implemented with install-able driver messages.

The pen driver is a 16-bit installable driver in the Microsoft Windows 95 operating system. All communication with an installable driver is through driver messages. A 16-bit application can send a message to the pen driver with the Windows API **SendDriverMessage** function.

Because a call to **SendDriverMessage** must originate from a 16-bit virtual machine, 32-bit applications cannot use **SendDriverMessage** to send messages directly to the pen driver. To communicate with the pen driver, a 32-bit application must provide its own 16-bit dynamic-link library (DLL) to "thunk" calls to **SendDriverMessage**.

The installable portion of the pen driver may not exist in future versions of Win-dows. For this reason, an application should not query for device information directly from the driver unless necessary. Instead, an application should get hardware information about an **HPENDATA** object by calling **GetPenDataInfo** or **GetPenDataAttributes**. These functions can apprise an application of various hardware characteristics (such as sampling rate) current when collecting the **HPENDATA**.

Opening the Pen Driver

Before sending a driver a message, an application must first obtain a handle to the driver with the Windows **OpenDriver** function. The following code demonstrates this:

```
HDRVRhDrvPen;
     .
     .
     .
hDrvPen = OpenDriver( "pen", NULL, NULL );
if( hDrvPen == NULL )
{
     // The pen driver does not exist.
     // Either display an error message and exit,
     // or continue to function as a pen-unaware application.
}
```

As an example of how to send the driver messages, the following code uses the pen driver message DRV_SetPenSamplingRate to set the sampling rate to 200 points per second. A later segment of code then queries the driver to get relevant pen infor-mation.

```
WORD wOldRate;

wOldRate = SendDriverMessage( hDrvPen,                 // Driver handle
                              DRV_SetPenSamplingRate,  // Message
                              200,                     // New rate in Hz
                              NULL );                  // Not applicable
     .
     .
     .
// Get information about the pen driver
PENINFO  pi;
BOOL     PenHardwareExists;

fPenHardwareExists = SendDriverMessage( hDrvPen,
                                        DRV_GetPenInfo,
                                        (DWORD)(LPPENINFO)&pi,
                                        NULL );
```

When finished, an application must close the handle to the installable driver with the **CloseDriver** function, as shown here:

```
CloseDriver( hDrvPen, NULL, NULL );
```

Pen Driver Return Values

All the values that can be returned by the Pen Driver in response to a message are listed in the table below. These return value constants are defined in PENDRV.H.

Return Value	Description
DRV_SUCCESS	The message request was completed successfully.
DRV_FAILURE	The message request was attempted, but was not completed successfully.
DRV_BADPARAM1	The message request was not attempted because the first parameter in the message was invalid.
DRV_BADPARAM2	The message request was not attempted because the second parameter in the message was invalid.
DRV_BADSTRUCT	The message request was not attempted because a message parameter that must point to a structure was not pointing to a valid structure of the required type. For example, a message parameter that should point to a **PENINFO** structure does not point to a writeable block of memory large enough to contain a **PENINFO** structure.

Pen Driver Messages

The following table describes the pen driver messages an application can use, the parameters that must be provided with each message, and the message return values.

Pen driver message	Meaning	Parameters	Return value
DRV_Configure	Requests the driver to display a configuration dialog box.	*lParam1* LOWORD is the handle to a window that will own the dialog box; *lParam2* is 0.	Returns DRV_SUCCESS or, if the window handle is invalid, returns DRV_FAILURE.
DRV_GetCalibration	Instructs the driver to fill a **CALBSTRUCT** structure with the current calibration settings, including size and off-set values.	*lParam1* is a far pointer to a **CALBSTRUCT** structure to be filled; *lParam2* is 0.	Returns DRV_SUCCESS or DRV_FAILURE if the installed tablet does not support re calibration. DRV_BADPARAM1 is returned if *lParam1* is invalid.
DRV_GetName	Reports the name of the pen hardware.	*lParam1* LOWORD is the length of the name buffer; *lParam2* is a LPSTR pointer to the name buffer.	If successful, returns the number of characters copied into the name buffer, which is the number of characters in the driver name plus 1 (the string null termina-tion character is copied into the buffer). Otherwise, returns DRV_BADPARAM1 because provided buffer length is insufficient or DRV_BADPARAM2 because the provided memory block is not a writeable block of the

			specified size.
DRV_GetPenInfo	Fills in the **PENINFO** structure pointed to by the message's *lParam1* parameter with the current pen parameters. If *lParam1* is set to NULL, the driver checks for the presence of a pen tablet only.	*lParam1* is a far pointer to a **PENINFO** structure to be filled; *lParam2* is 0.	Returns DRV_SUCCESS or DRV_FAILURE if no pen device is currently connected. DRV_BADPARAM1 is returned if the provided block of memory is not writeable or not large enough to contain a **PENINFO** structure.
DRV_GetPenSamplingRate	Returns the current sampling rate.	*lParam1* is 0; *lParam2* is 0.	LOWORD contains the current sampling rate.
DRV_GetVersion	Reports the Pen API version number the driver supports and the pen packet size.	*lParam1* is 0; *lParam2* is 0.	HIWORD contains the pen packet size in bytes. Within the LOWORD, HIBYTE contains the minor version number, LOBYTE is the major version number.
DRV_PenPlayBack	Sends an array of recorded pen packets to the driver, which the driver then sends back to the system as though receiving the packets in real time from the device.	*lParam1* is a far pointer to an array of pen packets to be played through the system; *lParam2* LOWORD is the number of pen packets to play back, HIWORD is 0.	If successful, returns 1; if failure, returns a value less than 1.
DRV_PenPlayStart	Informs the pen driver of the format of the pen packet data it will receive after it receives the next DRV_PenPlayBack message. The	*lParam1* is a far pointer to a DWORD that is updated when the driver is done playing back pen packets; *lParam2* is either	Returns DRV_SUCCESS or DRV_FAILURE if the pen system is already in playback mode.

	driver does not begin sending pen packets into the system until it receives the DRV_PenPlayBack message.	0 or 1 and specifies the format of the pen packets passed to the driver (0 means version 1.0 packets and 1 means version 2.0 packets).	
DRV_PenPlayStop	Forces the driver out of playback mode.	*lParam1* is 0; *lParam2* is 0.	Always returns DRV_SUCCESS (the pen driver was in play-back mode and the playback mode was successfully stopped or the pen driver was already out of playback mode).
DRV_Query	Returns whether or not the version 2.0 pen driver supports the specified message.	*lParam1* LOWORD specifies the message queried about; *lParam2* is 0.	Returns DRV_SUCCESS if the message is supported or DRV_FAILURE if it is not.
DRV_QueryConfigure	Returns whether or not the driver can provide a configuration dialog box.	*lParam1* is 0; *lParam2* is 0.	Returns DRV_SUCCESS if the driver provides a configuration dialog box or DRV_FAILURE if it does not.
DRV_SetCalibration	Sets the tablet calibration.	*lParam1* is a far pointer to a **CALBSTRUCT** structure that describes the new calibration parameters the pen driver must use; *lParam2* is 0.	Returns DRV_SUCCESS or DRV_FAILURE if the tablet does not support calibration. DRV_BADPARAM1 is returned if *lparam1* is not a pointer to a **CALBSTRUCT** structure.
DRV_SetPenSamplingDist	Sets the minimum pen sampling distance. Successive points less than the given	*lParam1* HIWORD is 0; *lParam1* LOWORD is a new sampling distance;	If successful, HIWORD contains 0; LOWORD contains the previous sampling

		distance do not generate new points. The distance is defined in raw tablet coordinates as the maximum of the change in x and y. The default distance is 0, which means that all pen events generate new events. A pen driver does not have to simulate non zero sampling distances. An application must use the DRV_GetPenInfo driver message to determine the actual sampling distance set. Pressing or releasing a pen barrel button gen-erates a new event, even if the pen does not move.	*lParam2* is 0.	distance. Otherwise, returns DRV_FAILURE.
	DRV_SetPenSamplingRate	Sets the pen sampling rate in samples per second.	*lParam1* HIWORD is 0; *lParam1* LOWORD is a new sampling rate; *lParam2* is 0.	If successful, HIWORD contains 0; LOWORD contains the previous sampling rate. Otherwise, returns DRV_FAILURE.

The calibration driver messages use the **CALBSTRUCT** structure defined as follows:

```
typedef struct
{
     int  wOffsetX;
     int  wOffsetY;
     int  wDistinctWidth;
     int  wDistinctHeight;
```

```
} CALBSTRUCT, FAR * LPCALPSTRUCT;
```

The **wOffsetX** and **wOffsetY** members are the amount in tablet coordinates that need to be added to the x- and y-coordinate values returned by the hardware for proper calibration. The **wDistinctWidth** and **wDistinctHeight** members have the same meaning as in the **PENINFO** structure.

A

action handle
A small icon provided in edit controls that facilitates an editing tasks such as dragging or insertion.

B

baseline
An imaginary horizontal line on which handwritten text rests. Analogous to the lines of lined notebook paper.

C

comb

A form of writing guide, such as those used in many common forms and questionnaires, consisting of a horizontal line with spaced tick marks. The guide gets its name from the resemblance of the tick marks to the teeth of a comb. The user writes in a comb guide with individual characters separated by the tick marks.

confidence level

A value assigned by a recognizer to indicate its degree of certainty in the results of a recognition. For example, in recognizing the word "clear," a recognizer assigns a confidence level to each of the five letters and can also assign a confidence level to the entire word. Word lists and dictionaries can influence confidence levels. *See also* recognizer.

D

dictionary

A list of words or phrases private to a recognizer. A recognizer can use its dictionaries to verify recognition guesses, as directed through the **EnableSystemDictionary**

G

gesture

A predefined pen motion that signifies some desired action, such as a "lasso" to select or an X to delete. *See also* lasso.

H

hook

A callback function provided by an application that receives certain data before the normal recipient of the data. The hook function can thus examine or modify the data before passing it on.

hot spots

Critical points on symbols, particularly gestures, identified by a recognizer during recognition. *See also* gesture, recognizer, symbol.

ink

(1) A trail of colored pixels left on the screen that marks the path of the pen's motion. (2) Input data generated by the moving pen; pen data.

inkset

An object that maps data points to time intervals. The points recorded in an inkset object may describe one or more strokes. *See also* stroke.

irreversible compression

A data-compression technique that produces a high degree of compression, but at the cost of lost information. After uncompression, the pen data can be redisplayed but recognition accuracy may be reduced. *See also* reversible compression.

L

lasso

A gesture formed by circling a section of text or other displayed data. *See also* gesture.

lens

A standard Windows pen interface dialog box that offers an on-screen keyboard or letter guides and is used for entering and editing text.

lossless compression
See reversible compression.

lossy compression
See irreversible compression.

o

OEM data

Data about pen pressure, angle, height, and other aspects of pen input that is collected in addition to data points. The specific OEM data collected depends on the hardware and the data it reports.

on-screen keyboard

(1) An image of a keyboard displayed on the screen. (2) The applet (Screen Keyboard or SKB) that displays the image. The user "types" on the on-screen keyboard by pressing the pen on the desired keys, as though typing on a real keyboard.

P

packet

A collection of pen data sent by the pen driver at a frequency determined by the sampling rate. Each packet contains the current coordinates of the pen, the time, and, optionally, other information. Collectively, the packets represent a digitized history of the pen's movement. *See also* sampling rate.

pen collection mode

The system state in which pen movement generates data, instead of being interpreted as mouse movement. *Also known as* input session.

pen-down stroke

Data points collected while the pen is in contact with the tablet. Together, these points comprise a stroke. *See also* pen state, pen tip transition, pen-up points, proximity stroke.

pen packet
See packet.

pen state

The condition of the pen relative to the tablet surface – either up or down, depending on whether the pen is in contact with the tablet.

pen tip transition

The act of touching a pen to or lifting the pen from the tablet surface. The former begins a pen-down stroke, while the latter ends a pen-down stroke and begins a pen-up stroke. *See also* stroke, proximity stroke.

pen-up stroke

Data points received when the pen is not in actual contact with the tablet, but near enough for the tablet to sense movement. Together, these points comprise a proximity stroke. *See also* pen-down points, pen state, pen tip transition, proximity stroke.

proximity stroke

A stroke formed while the pen is near but not on the tablet surface. Also called "pen-up stroke." Not all pen tablets can detect proximity strokes. *See also* pen tip transition, pen-up points, stroke.

R

real time

In the context of pen-based computing, real time means "while the pen is moving."

recognition function

One of the 43 Pen Application Programming Interface (API) functions exported by a recognizer dynamic-link library. *See also* recognizer.

recognizer

A dynamic-link library of functions that collectively serve to recognize ink data as characters, numerals, words, foreign script, or other symbols.

reentrancy

The condition in which a function is interrupted during execution and restarted from its beginning in response to another caller.

reversible compression

A data-compression technique that loses no information, so data can be redisplayed and recognized with no loss of accuracy after uncompression. *See also* irreversible compression.

S

sampling rate

The frequency at which the pen driver sends packets to the pen system. A typical sampling rate is 100 - 120 packets per second, but does not necessarily equal the rate of hardware interrupts generated by the pen tablet. *See also* packets.

Screen Keyboard
See on-screen keyboard.

SKB

See on-screen keyboard.

stroke

The pen data generated between two pen tip transitions. For example, when the pen touches the tablet (pen down), all data generated as the pen moves comprises a pen-down stroke until the pen leaves the tablet. If the tablet can sense proximity strokes, the pen movement above the tablet surface forms a separate pen-up stroke until the pen either leaves the tablet's range of sensitivity or touches down again. Thus, in noncursive printing, the letter "c" is formed as a single stroke while the letter "E" requires several pen-down strokes separated by pen-up strokes. *See also* pen-down points, pen tip transitions, pen-up points, proximity strokes.

symbol

An element interpreted by a recognizer from raw pen data. For example, the default system recognizer sees an individual letter or numeral as a symbol. A word is thus a string of symbols. A recognizer for cursive writing, however, may see an entire word as a single symbol without distinguishing each letter of the word. *See also* recognizer.

symbol correspondence

A map of the ink data that forms a recognized symbol. *See also* symbol element.

symbol element

An **SYE** structure containing a symbol value, its confidence level, and an identification of the ink data that forms the symbol. *See also* confidence level, symbol value.

symbol value

A numerical value that represents a recognized character or set of characters. A symbol value is internal to the recognizer and by itself has little meaning to the application. To translate a symbol value to a character such as a letter or numeral, an application must call the **SymbolToCharacter** function.

T

target
A window or writing area that receives pen input data.

trainer

An application that trains a recognizer. The trainer application may operate in the background, which is known as "passive training," or be activated by the user, which is "active training." *See also* recognizer, training.

training

The process used to update a recognizer's database so that it better reflects the individual style and writing characteristics of a particular user, thus increasing the recognizer's accuracy in handwriting interpretation. *See also* recognizer, trainer.

Programmer's Guide to Pen Services for Microsoft® Windows® 95 (Addendum)

This addendum to the *Programmer's Guide to Pen Services for Microsoft Windows 95* contains updated information intended for international application development, particularly for Japanese pen-based applications. The changes listed in this addendum are divided by chapter and appendix.

Introduction

- The last sentence of first paragraph should read as follows:

  Following the reference, a number of appendixes provide information about the differences between versions 1.x and 2.0 of the Pen API, the 32-bit pen services, and more.

Organization

- In the table describing the parts of the book, note that the title of Appendix A should refer to Versions 1.x and 2.0 rather than 1.0 and 2.0.

Books and Articles for Further Reading

- The description of Ray Duncan's article, "Power Programming," in *PC Magazine*, should refer to version 1.x of the Pen API, not version 1.0.

Chapter 1 Overview of the Pen Application Programming Interface

- Note the version change in the final introductory paragraph:

  The architecture of version 2.0 of the Pen API remains similar to version 1.x, but its style and design differ considerably. Even if you have worked with version 1.x, you should read this chapter to understand the shift in programming philosophy in version 2.0.

Architecture of the Pen API

- Note the version change in this paragraph from the "Windows" section.

  In version 1.x of Pen Windows, the application was required to call **RegisterPenApp** in order to tell the system to convert all edit controls to handwriting edit (hedit) controls. With Pen API version 2.0, however, this is not necessary; all edit controls in applications are automatically converted. If the application is version-stamped as a Windows 95-based application, the conversion is automatic; otherwise, applications version-stamped as Windows 3.1-based applications require the call to **RegisterPenApp** that was required for Pen Windows, version 1.x.
- Information about handwriting recognizers has been added to this paragraph in the "Recognizer" section.

  Although many recognizers may be available to an application, only one serves as the system default recognizer. This is the recognizer that Windows automatically installs and calls by default. To use other recognizers, an application must first specifically install them. (For information about how to install multiple recognizers, see Chapter 5, "The Recognition Process.") The Microsoft Handwriting Recognizer (GRECO.DLL in US, RODAN.DLL in Japan) is provided as the default system recognizer on most OEM tablet installations of Microsoft pen services. GRECO.DLL recognizes all European letters, numerals, and punctuation, with emphasis on English, French, and German. RODAN.DLL recognizes Kanji, Kana, and other Japanese characters. An application can set up a different system recognizer by identifying the new file in the Windows registry. Appendix A explains how to set up a new default recognizer.
- The following section was added at the end of the "Recognizer" section.

IME (Input Method Editor, for converting Kana to Kanji)

In Japanese Pen Services 1.1 (for Windows 3.1) special extensions to the system IME were required to convert handwritten characters to Kanji. For Windows 95, the IME has become standardized as a system DLL, with a new, extended API set. No special API set is required for use with handwriting input.

Although the IME shipped with Windows 3.1 for Pen may be used in Windows 95, it cannot be used to convert handwritten characters into Kanji. Some of the original 1.1 Pen APIs used for Kana to Kanji conversion have been updated, and remain backwardly compatible, and some new APIs have been added. For more information, see Chapter 10, "Pen Application Programming Interface Functions."

Chapter 2 Starting Out with System Defaults

Pen-Unaware Applications

- This paragraph was added as the last in the section.

  In Japan, these applets are combined into a single application called the Data Input Window (PENSKJ.EXE). In addition, an applet to set system recognition priorities (Tool Palette, PENTPJ.EXE) and an applet to train Japanese characters (Trainer, PENTRJ.EXE) are also provided.

Chapter 3 The Writing Process

- The second introductory paragraph has been changed, as follows:

  The writing process includes the various ways a user can write input to a pen-based application. These involve not only writing words and scribbling figures with a pen, but also gesturing with predefined pen movements and tapping an on-screen keyboard or Data Input Window.
- The following Note has been added as the final item in "The hedit Control" section.

Note The Microsoft Japanese Handwriting Recognizer (RODAN.DLL) does not support "free input" into hedit controls. An application should either use a BEDIT control or provide access to the Data Input Window—with a Lens button, for example.

- Figure 3.1 has been renumbered as 3.1a, and a new piece of art (Figure 3.1b) has been added.

{ewc msdncd, EWGraphic, bsd23556 0 /a "SDK.BMP"}

Japanese characters typically use large, separated boxes, often with a small cross at the center of each as a writing aid:

{ewc msdncd, EWGraphic, bsd23556 1 /a "SDK.BMP"}

- The version number in the first sentence of the third paragraph in the "bedit Control Messages"section has been changed.

  The EM_LIMITTEXT message deserves special mention because it has changed slightly from version 1.x of the Pen API.
- In the same section, a final bulleted item has been added to the ending list.
  - If limiting the text splits a double-byte character, the whole character is discarded.

The On-screen Keyboard

- The following paragraph has been added after the second paragraph.

  In Japan, the **ShowKeyboard** API has been replaced by the **CorrectWritingEx** function, which displays the Data Input Window.

Chapter 4 The Inking Process

- Note the version change in this paragraph from the "Stroke Headers" section of "Data Within an HPENDATA Object."

  As Figure 4.1 shows, a stroke header prefaces each collection of pen coordinates that make up a single stroke. Note that the structure of the stroke header in version 2.0 of the Pen API is different from the stroke header of version 1.x, since the stroke header now consists of a variable-length array instead of the **STROKEINFO** structure used in version 1.x. The current **STROKEINFO** structure is, nevertheless, compatible with version 1.x stroke headers.
- Note the version change in the following Note from the "Creating an HPENDATA Object" section.

Note It is highly recommended that you use only the functions from version 2.0 of the Pen API. Although API from version 1.x are included for backward compatibility, it is not guaranteed that they will be supported in future versions of the Pen API.

Chapter 5 The Recognition Process

Using the HRC Functions

- In the section "Creating the HRC," a bulleted item has been added following the third paragraph. Also, the paragraph after the bulleted list contains a change.

  Either or both arguments can be NULL. Giving NULL as the first argument creates a new **HRC** with default settings. The next section, "Configuring the HRC," describes the default parameters, which include the following settings:

  - Recognition ends after a brief period of inactivity or when the user taps outside the target window.
  - The target window does not use guides.
  - The recognizer returns only its best guess without alternative guesses.
  - In Japan, recognition priority defaults to the settings specified by the Tool Palette applet, if it is active; otherwise the recognizer default setting is used.

  Giving NULL as the second argument binds the **HRC** to the system default recognizer. Microsoft Windows sets the supplied file GRECO.DLL as the system default recognizer (RODAN.DLL in Japan), specified in the Microsoft Windows 95 system registry. Refer to Appendix A for an explanation of how to change the default to another recognizer.
- The following information has been added immediately before the "Gesture" section.

ALC_KANJISYSMINIMUM	Minimum set of characters needed for Japanese system recognizer. Same as ALC_SYSMINIMUM | ALC_HIRAGANA | ALC_KATAKANA | ALC_JIS1. (Japanese version only.)

  If an application does not specify alphabet configuration either through an existing **HRC** model or by calling the function **SetBoxAlphabetHRC** or **SetAlphabetHRC**, Windows assumes ALC_SYSMINIMUM (or ALC_KANJISYSMINIMUM) as the default alphabet configuration. For a complete list of ALC_ values, including Japanese-specific ones, see Chapter 13, "Pen Application Programming Interface Constants."

Alphabet Priority

The **SetAlphabetPriorityHRC** function provides a hint to the recognizer about which alphabet the user intends to use while inputting handwritten characters. Some written symbols have more than one interpretation, depending on alphabet (this is especially true in Japanese, which uses multiple alphabets). A crosslike character like the English plus sign (+) could also be interpreted as a lower case "t", katakana "na" or "me", or Kanji "juu" (the number 10), for example. Moreover, there are both SBCS and DBCS versions of English and katakana characters, corresponding to half-and full-pitch widths, so it's useful for a user or application to be able to pre-specify widths and character set preferences to minimize recognition errors.

Applications may use the **SetAlphabetPriorityHRC** API to set width and alphabet priority for HRC scope, or the **SetPenAppFlags** API to set width (DBCS or SBCS) preference for application scope. The Tool Palette application may be used to set system-wide priorities. Any priority set into an HRC supersedes application priority, which supersedes system priority as set (and if set) by the Tool Palette; in the event of not Tool Palette, the recognizer's default priority is used. The ALC_GLOBALPRIORITY bit will be set in all default HRCs, unless it is explicitly cleared with **SetAlphabetPriorityHRC**. To specify no priority, as opposed to default priority, an application should clear the ALC_GLOBALPRIORITY bit and set ALC_NOPRIORITY.

- The following constant has been added to the "Gesture" section.

GST_KKCONVERT	Kana-Kanji convert. (Japanese version only)

- In the "Unboxed Recognition" section, the following paragraphs have been changed.

  An **HRCRESULT** object does not contain a normal text (ASCII for English) string representation of a guess. This is not possible since a guess might be made up of a gesture, shape, or other entity that has no text equivalent. Instead, an **HRCRESULT** contains a string of *symbol values*, which are 32-bit numbers type-defined as **SYV**.

  Symbol values can represent geometric shapes, gestures, letters of the alphabet, Japanese kanji characters, musical notes, electronic symbols, or any other symbols defined by the recognizer. The Pen API provides the function **SymbolToCharacter** to convert the null-terminated symbol string in **HRCRESULT** to a normal text string.
- In the section "Getting Results from the RCRESULT Structure," note the version change in the first sentence of this paragraph.

  The **RCRESULT** structure applies only when an application calls either of the version 1.x recognition functions, **Recognize** or **RecognizeData**. In this case, the system sends a WM_RCRESULT message to the application. The *wParam* of this message contains a REC_ submessage that indicates why recognition ended. The *lParam* of WM_RCRESULT points to an **RCRESULT** structure, which contains all the results of the recognition.
- Note the change to "text string" (formerly "ASCII string") in this paragraph.

  The **RCRESULT** structure identifies the recognizer's "best guess," which is the guess in which the recognizer places the most confidence. With this information, an application can conveniently retrieve a text string of the best guess by calling **SymbolToCharacter**:

Chapter 6 Design Considerations

Recognition: Use and Misuse

- In the section "bedit Is Better Than hedit," the following third paragraph has been added.
  In Japan the bedit is almost always used for input, either directly, or via the Data Input Window.
- In the section "Provide Easy Access to the On-screen Keyboard," the following sentence has been added after the first paragraph.
  In Japan, this information is served by the Data Input Window.

Guidelines for Applications

- The "Word Processor" section has been changed.

  Although the pen does not reasonably serve to create a word-processor document for English or European languages, it can serve well for small editing tasks on an existing document, such as cut-and-paste operations, formatting changes, rewriting small amounts of text, and navigation (scrolling)....

  - In Far Eastern countries such as Japan, Korea, and China, where ideographic characters such as Kanji are used, a full-featured pen-input word processing application would greatly simplify input of text.

Chapter 7 A Sample Pen Application

Overview of PENAPP

- In the first bulleted item following the third paragraph, the phrase "and it supports free input" has been added and the word "ANSI" has been deleted.

Window Procedures

- The following code has been changed in the "InputWndProc" section.

```
#define   MAX_GUESS   5     // Maximum number of guesses
#define   MAX_CHAR    20    // Maximum number of characters per guess

// Global Variables ************************************************
HRCRESULT   vrghresult[MAX_GUESS];        // Array of results
SYV    vsyvSymbol[MAX_GUESS][MAX_CHAR];   // Array of symbol strings
int         vcSyv[MAX_GUESS];             // Array of string lengths
.
.
.

LRESULT CALLBACK      InputWndProc(
                           HWND hwnd,      // Window handle
                           UINT message,   // Message
                           WPARAM wParam,  // Varies
                           LPARAM lParam ) // Varies
{
   LONG            lRet = 0L;      // Initialize return code to FALSE
   HRC           hrc;              // HRC object
   HDC           hdc;
   PAINTSTRUCT   ps;
   DWORD         dwInfo;
   int           i, cGuess;

   switch (message)
   {
         .
         .
         .
case WM_LBUTTONDOWN:
//
// Two possibilities exist: user is using mouse or the pen.
// The latter case indicates the user is starting to write.
//
dwInfo = GetMessageExtraInfo();
if (IsPenEvent( message, dwInfo ))
{
    if (DoDefaultPenInput( vhwndInput, (UINT)dwInfo ) == PCMR_OK)
        lRet = TRUE;
    else
        lRet = DefWindowProc( hwnd, message, wParam, lParam );
}
break;

case WM_PENEVENT:
    switch (wParam)
    {
        case PE_GETPCMINFO:
        //
        // If using SREC recognizer, ensure session ends
        // on pen-up.
```

```
        //
        if (viMenuSel == miSample)
            ((LPPCMINFO) lParam)->dwPcm |= PCM_PENUP;
        lRet = DefWindowProc( hwnd, message, wParam, lParam );
        break;

case PE_BEGINDATA:
    //
    // Action based on current menu selection:
    //
    // 1) If currently using sample recognizer, create an HRC
    // for it and specify it in the TARGET structure pointed
    // to by lParam.  This tells DoDefaultPenInput to use
    // the sample recognizer rather than the system default.
    //
    // 2) If displaying mirror image of ink, create an
    // HPENDATA for it and specify it in the TARGET structure
    // pointed to by lParam.  This tells DoDefaultPenInput to
    // collect data into the HPENDATA block instead of
    // passing it to a recognizer.
    //
    // 3) If using the default recognizer, pass to DefWindowProc,
    // which sets the maximum number of guesses to 1 and may
    // or may not require any guide information. The Japanese
    // default recognizer (RODAN.DLL) does require a guide
    // for character recognition. The following code shows
    // how to access the HRC that DefWindowProc creates,
    // reset the number of guesses to MAX_GUESS, and set
    // up the client rectangle of the input window as a
    // single guide.
    //
    if (vhpendata)
    {
        DestroyPenData( vhpendata );
        vhpendata = 0;
    }

    switch (viMenuSel)
    {
        case miSample:
            hrc = CreateCompatibleHRC( NULL, vhrec );
            if (hrc)
            {
                ((LPTARGET) lParam)->dwData = hrc;
                lRet = LRET_HRC;
            }
            break;

        case miMirror:
            vhpendata = CreatePenData( NULL, 0,
                                       PDTS_HIENGLISH, 0 );
            if (vhpendata)
            {
                ((LPTARGET) lParam)->dwData = vhpendata;
                lRet = LRET_HPENDATA;
```

```
            }
            break;

        case miSystem:
            lRet = DefWindowProc( hwnd, message,
                                  wParam, lParam );
            //
            // On return, lParam->dwData points to HRC.
            // Use it to reset max number of guesses.
            //
            SetMaxResultsHRC( ((LPTARGET) lParam)->dwData,
                              MAX_GUESS );
            {
            GUIDE guide;
            RECT rc;

            GetClientRect(vhwndInput, &rc);
            ClientToScreen(vhwndInput, (LPPOINT) &rc.left);
            ClientToScreen(vhwndInput,    (LPPOINT) &rc.right);
            guide.xOrigin = rc.left;
            guide.yOrigin = rc.top;
            guide.cxBox = rc.right - rc.left;
            guide.cyBox = rc.bottom - rc.top;
            guide.cxBase =    0;
            guide.cyBase =    guide.cyBox;
            guide.cHorzBox = 1;
            guide.cVertBox = 1;
            guide.cyMid = guide.cyBox / 2;
            SetGuideHRC(((LPTARGET) lParam)->dwData, &guide, 0);
            }
            break;
}
    break;

case PE_ENDDATA:
//
// DefWindowProc will destroy vhpendata, so if collecting
// mirror image, don't let DefWindowProc handle message
//
if (viMenuSel != miMirror)
        lRet = DefWindowProc( hwnd, message, wParam, lParam );
    break;

case PE_RESULT:
//
// At end of input, collect recognition results (if any)
// into symbol strings.  DoDefaultPenInput generates the
// PE_RESULT submessage only when using a recognizer.
// The lParam contains the HRC for the recognition process.
//
// NOTE:
//    Do not destroy HRC, even after getting results!
//    DefWindowProc takes care of destroying the object.
//
```

```
// Collect pen data for DrawRawData
vhpendata = CreatePenDataHRC( (HRC) lParam );

// Initialize array to zero
for (i = 0; i < MAX_GUESS; i++)
    vcSyv[i] = 0;

// Get number of guesses available
cGuess = GetResultsHRC( (HRC) lParam,
                    GRH_ALL,
                    (LPHRCRESULT) vrghresult,
                    MAX_GUESS );

// Get guesses (in vsyvSymbol) and their lengths (invcSyv)
if (cGuess != HRCR_ERROR)
    for (i = 0; i < cGuess; i++)
        vcSyv[i] = GetSymbolsHRCRESULT( vrghresult[i],
                            0,
                            (LPSYV) vsyvSymbol[i],
                            MAX_CHAR );

// Destroy the HRCRESULTS
for (i = 0; i < cGuess; i++)
    DestroyHRCRESULT(vrghresult[i]);

                break;
                .
                .
                .
        default:
            lRet = DefWindowProc( hwnd, message, wParam, lParam );

    }                   // End switch (message)
    return lRet;
}
```

- In the "DisplayGuesses" section of the "InfoWndProc" section, the following code has changed.

```
VOID        DisplayGuesses( HDC hdc )               // DC handle
{
    TEXTMETRIC  tm;
    int            nX, nY;                           // Text coords
        .
        .
        .
    for (i = 0; i < MAX_GUESS; i++)
    {
        if (vcSyv[i] > 0)
        {
            SymbolToCharacter( (LPSYV) vsyvSymbol[i],
                            vcSyv[i],
                            (LPSTR) szText,
                            (LPINT) &cChar );
            for (nLen = 0; cChar > 0; cChar--)
                nLen += IsDBCSLeadByte(*(szText + nLen)) ? 2 : 1;
```

```
            TextOut( hdc, nX, nY, (LPSTR) szText, nLen );
            nY += tm.tmExternalLeading + tm.tmHeight;
        }
    }
}
```

Chapter 8 Writing a Recognizer

How a Recognizer Works

- Version information has been changed in the first paragraph.

  The Pen API specifies the following functions for initializing, modifying, and closing down the recognizer. Note that, in version 2.0 of the Pen API, the required function **ConfigRecognizer** handles all initialization and configuration tasks. The other initialization functions are obsolete in version 2.0 and should only be included in a recognizer if it is expected to work with older applications that work with a version 1.x recognizer (see the Microsoft Pen Windows, version 1.x documentation for descriptions of these functions).

  Note that the table following the first paragraph should refer to version 1.x rather than 1.0.
- In the "HRCRESULT Section," note that the **GetAlternateWordsHRCRESULT** function is not supported.
- In the "Training" section, note the version changes in the following paragraph and in the table it introduces.

  The following table lists the functions that a recognizer with training capabilities can export. Only **TrainHREC** is used by version 2.0 Pen API. The other functions are obsolete in version 2.0 and should be included in a recognizer only if it is expected to work with older applications that work with a version 1.x recognizer (see the Microsoft Pen Windows, version 1.x documentation for descriptions of these functions).
- In the "Processing Raw Data" section of the "Interpreting Input" section, note the additional .DLL in the first sentence of the second paragraph.

  The Microsoft Handwriting Recognizer (GRECO.DLL or RODAN.DLL) processes only coordinate data.
- In the section "Segmentation of Symbols," following Figure 8.1, the final sentence has been changed.

  The Pen API places few restrictions on the recognizer. At a minimum, however, a default recognizer that supports free (unboxed) input must be able to recognize discrete characters because many applications do not use boxed input.
- In the "Results Messages" section of the "Returning Results" section, note the version change in the second bulleted item.
- In the section "The RCRESULT Structure" in the "Returning Results" section, the note should refer to version 1.x, not 1.0. Also in this section, the "Location and Position of the Input" section, the final sentence in the first bulleted item has been changed.

  The Microsoft Handwriting Recognizer (GRECO.DLL or RODAN.DLL) sets **nBaseLine** to 0.

Writing a Recognizer

- In the "DestroyHRC" section, the code has been changed.

```
int WINAPI    DestroyHRC( HRC hrc )
{
    LPHRCinternal  lphrc = (LPHRCinternal) hrc;  // Pointer to HRC

    if (GlobalUnlock( lphrc->hglobal ) || GlobalFree(lphrc->hglobal))
        return HRCR_ERROR;
    else
        return HRCR_OK;
}
```

Chapter 9 Summary of the Pen Application Programming Interface

Pen API Functions

- Note the version change in the table of function categories.

Obsolete	Obsolete functions of version 1.x maintained by version 2.0 only for compatibility reasons.

- In the "List of Pen API Functions" section, the descriptions of several functions have been changed.

ShowKeyboard	Displays or hides the on-screen keyboard. (Not supported in Japan)
GetAlternateWordsHRCRESULT	Gets alternative guesses made by a recognizer. (Not supported in Japan)
CorrectWriting	Displays lens or Correct Text dialog box to allow user to correct text. (In Japan, this is wrapper for **CorrectWritingEx**)
KKConvert	Activates the Kana-to-Kanji converter. (Japanese version only.)
StartPenInput	Begins collecting into an internal buffer ink data generated by the moving pen. See also the descriptions of **StartInking** and **DoDefaultPenInput**.
Recognize	Begins recognition for a version 1.*x* recognizer.
RecognizeData	Delayed recognition for a version 1.*x* recognizer.
SetRecogHook	Installs and removes a recognition hook in version 1.*x.* Superseded by **SetResultsHookHREC**.

Pen API Structures

- In the "List of Pen API Structures" section, the descriptions of several structures have been changed.

RC	Various information about the recognition context used by version 1.*x* recognition functions.
RCRESULT	Results of recognition initiated through a version 1.*x* recognition function.

Pen API Constants

- The following items have been either redescribed or added to the table of constants.

CWX_	Options for **CorrectWritingEx**.
CWXA_	Options for **CorrectWritingEx**.
CWXK_	Options for **CorrectWritingEx**.
CWXKS_	Options for **CorrectWritingEx**.
CWXR_	Return types for **CorrectWritingEx**
GPR_	Options for **GetPenResource**.
REC_	Return codes from a version 1.*x* recognizer.

- In the second paragraph, the phrase "for example, 1.0 or 2.0" should read "for example, 1.x or 2.0."
- The CWR_KKCONVERT constant has been removed from the *dwFlags* parameter of the **CorrectWriting** function. Also, in the same function, the final sentence has been changed as follows:

  Note that in the Japanese version, **CorrectWriting** is supported but internally calls **CorrectWritingEx**, which opens a Data Input Window.
- The first sentence of the **CorrectWritingEx** funtion has been changed:

  Sends text to the Japanese Data Input Window to allow the user to edit text. (Japanese version only.)

  In the description of the *lpText* parameter of **CorrectWritingEx**, the phrase "and gets modified by the user" has been appended to the second sentence.
- In the description of the cbText parameter, the second sentence should read as follows:

  If the source of the text is an edit control that is constrained by EM_LIMITTEXT, *cbText* should be set to the limiting size plus one.
- In the Example of CompactWritingEx, the second-to-last code block has been changed.

```
// don't update most-recently used settings for this one-shot:
cwx.wApplyFlags |= CWXA_NOUPDATEMRU;
cwx.ixkb = CWXK_QWERTY;
cwx.rgState[CWXK_QWERTY-CWXK_FIRST] = CWXKS_ROMAHAN;
cwx.dwFlags = CWX_NOTOOLTIPS | CWX_TOPMOST;    // no distractions
```

- In the **GetHotspotsHRCRESULT** function, the final sentence of the first Comments paragraph has been changed.

  The Microsoft Handwriting Recognizer (GRECO.DLL in US, RODAN.DLL in Japan), supports this function for gesture symbols only.
- In the **GetPenAppFlags** function, the three references to the **RegisterPenApp** function have been changed to **SetPenAppFlags**.
- The **GetPenResource** function has been added.

GetPenResource

The **GetPenResource** function retrieves a copy of a pen services resource. (Japanese version only.)

HANDLE GetPenResource(WPARAM *wParam* **)**

Parameters

wParam

Specifies the pen services resource for which to retrieve a handle. This may be one of the following:

Constant	Description
GPR_CURSPEN	Standard pen cursor.
GPR_CURSCOPY	Copy cursor.
GPR_CURSUNKNOWN	Unknown cursor.
GPR_CURSERASE	Erase cursor.
GPR_BMCRMONO	Monochrome Return bitmap.
GPR_BMLFMONO	Monochrome LineFeed bitmap.
GPR_BMTABMONO	Monochrome Tab bitmap.
GPR_BMDELETE	Delete bitmap.
GPR_BMLENSBTN	Lens buttonface bitmap.
GPR_BMHSPMONO	Hankaku space bitmap (Japanese version only).
GPR_BMZSPMONO	Zenkaku space bitmap (Japanese version only).

Comments

An application can use this function to get a copy of a cursor or bitmap used by pen services. It is the application's responsibility to destroy the object by calling either the **DestroyCursor** or **DeleteObject** Windows API.

Return Value

- This function returns a handle to an object, depending on the index specified by *wParam* if successful. Otherwise the return value is NULL.

- The following sentence has been added to the Comments section of the **SetPenAppFlags** function.

  If both RPA_DBCSPRIORITY and RPA_SBCSPRIORITY are specified, the RPA_SBCSPRIORITY is ignored.
- In the **TrainHREC** function, the following sentence has been appended to the Comments section.

  Training gestures depends on the recognizer. The Microsoft Handwriting Recognizer (GRECO.DLL in US, RODAN.DLL in Japan) does not support training for gestures.

Chapter 11 Pen Application Programming Interface Structures

- Note that throughout this chapter structures marked as version 1.0 should be marked as version 1.*x*.
- The final sentence of the introductory paragraph has been changed.

  The entry heading identifies the Pen API version, such as 1.*x* or 2.0, that supports the structure.
- In the CWX structure, the description of the CWXA_CONTEXT constant has changed.

Constant	Description
CWXA_CONTEXT	Use the *dwFlags* member to specify context. (topmost, tooltips, period, comma)

- Two CWX_ constant descriptions have been changed.

Constant	Description
CWX_EPERIOD	Specifies that the English period is to be used on some keys on the Data Input Window keypads. The Japanese period is used by default.
CWX_ECOMMA	Specifies that the English comma is to be used on some keys on the Data Input Window keypads. The Japanese comma is used by default.

- The CWXK_ROMAJI constant has been removed from the list of CWXK_ constants.
- The description of the CWXK_KANJI constant should read as follows:

  Kanji finder, which provides a method of specifying a Kanji character based on its strokes.
- The paragraph describing the **rgState** member of the **CWX** structure has been changed. The constant descriptions have also changed.

  ⋯and this member is ignored. The zero-based order is: 50-On Keyboard, QWERTY Keyboard, Numeric Keyboard, Stroke/Radical Finder, Code Finder, and Yomi Finder. On return, this member contains the updated states. Each element of the array may be CWXKS_DEFAULT, which causes the existing saved state to be used, or one of the following constant values, depending on the keyboard:

Constant	Description	Applicable Keyboards
CWXKS_HAN	Set Hankaku (single-byte) state.	Numeric
CWXKS_ZEN	Set Zenkaku (double-byte) state.	Numeric
CWXKS_ROMAHAN	Set Hankaku Romaji state.	QWERTY
CWXKS_ROMAZEN	Set Zenkaku Romaji .	QWERTY
CWXKS_HIRAZEN	Set Hiragana state.	50-On, QWERTY
CWXKS_KATAHAN	Set Hankaku Katakana state.	50-On, QWERTY
CWXKS_KATAZEN	Set Zenkaku Katakana state.	50-On, QWERTY

- The SKBINFO structure is not available in the Japanese version of the Pen API.

HE_KKCONVERT

Starts Kana-to-Kanji conversion. Submessage of WM_PENCTL. (Japanese version only.)

Parameters

wParam

HE_KKCONVERT.

lParam

Must be one of the following values:

Value	Meaning
HEKK_DEFAULT	The first time the conversion is specified, the selected or marked character string is replaced with the conversion result. The behavior for successive calls depends on the IME. For example, a candidate list may appear.
HEKK_CONVERT	The selected or marked character string is replaced with the conversion result; the conversion candidate list doesn't appear regardless of how many times conversion has been specified.
HEKK_CANDIDATE	The selected or marked character string is replaced with the conversion result; this causes the conversion candidate list to appear.
HEKK_DBCSCHAR	The SBCS characters (0x20 - 0x7E, 0xA1 - 0xDF) are replaced by their DBCS equivalents.
HEKK_SBCSCHAR	The DBCS characters in the selected or marked character string that have equivalent SBCS representations are replaced by their equivalent SBCS characters.
HEKK_HIRAGANA	The katakana characters (DBCS or SBCS) in the selected or marked character string are replaced with their hiragana equivalents.
HEKK_KATAKANA	The hiragana characters in the selected or marked character string are replaced with their DBCS katakana representation.

Return Value

Returns TRUE if there are no errors; otherwise, returns FALSE:

Comments

In this message, "marked character string" indicates the string that is marked for conversion. Text marked for conversion is indicated by a different selection color than that used for normal text selection in a standard text edit control. Available for bedits only.

HE_PUTCONVERTCHAR

Sends a character to the control and marks it for conversion. Submessage of WM_PENCTL. (Japanese version only.)

Parameters

wParam

HE_PUTCONVERTCHAR.

lParam

The low-order word contains the character code, which can be an SBCS or DBCS character.

Return Value

Returns TRUE if there are no errors; otherwise, returns FALSE.

Comments

Posting this message is similar to posting a WM_CHAR message to a bedit control, with the exceptions that the posted character also acquires the attribute of being a character marked for conversion in the Input Method Editor, and DBCS characters may be specified in the LOWORD of *lParam*. This submessage is available for bedits only.

WM_PENMISC

- The PMSC_KKCTLENABLE, PMSC_PENUICHANGE, and PMSC_SUBINPCHANGE subfunctions have been removed.

Chapter 13 Pen Application Programming Interface Constants

ALC_Alphabet Codes

Constant	Description
ALC_KANJISYSMINIMUM	Minimum set of characters needed for Japanese system recognizer. Same as ALC_SYSMINIMUM | ALC_HIRAGANA | ALC_KATAKANA | ALC_JIS1. (Japanese version only.)

RCD_ Writing Direction

- Note the Note: The RCD_ constants are provided only for compatibility with version 1.x the Pen API and will not be supported in future versions.

RCP_ User Preferences

- Note the Note: RCP_ constants are provided only for compatibility with version 1.x of the Pen API and will not be supported in future versions.

RCRT_ Results Type

- The RCRT_ALREADYPROCESSED constant description has been changed.

Constant	Description
RCRT_DEFAULT	Normal return type.

Appendix A Differences Between Versions 1.x and 2.0 of the Pen Application Programming Interface

Note the title change, from version 1.0 to 1.x. This change is reflected throughout this chapter and will not be mentioned here unless text has otherwise changed.

Improvements to the bedit Control

- The final paragraph of this section has been changed.

  Because Japanese and other Far Eastern languages may use double-byte characters, an application must specify double the number of bytes for EM_LIMITTEXT than applications expecting only single-byte characters (or halve the number of boxes). Note that even without explicitly limiting text with EM_LIMITTEXT, input into single-line, nonscrollable BEDIT controls is limited to the number of boxes available. For example, a 3-box postal code BEDIT allows only three characters, be they SBCS or DBCS. For more information about EM_LIMITTEXT, see "The bedit Control" in Chapter 3, "The Writing Process."

- In the section "The RC Structure," the table following the second paragraph has been changed.

RC member	Equivalent service in Pen API 2.0
lpUser	**GetPenMiscInfo** with PMI_USER. Cannot set new user in version 2.0.

Gestures

- The gesture table has been changed.

Gesture	Name	Action
{ewc ï¿½}	T-circle	Tab
{ewc ï¿½} or {ewc msdncd, EWGraphic, bsd23556 4 /a "SDK.BMP"} or	K-circle or symbol-circle (drawin Right to Left)	Kana to Kanji conversion

Action Handles

- A sentence has been added at the end of this section.
  In the Japanese version, action handles can be disabled with the Control Panel.

On-Screen Keyboard

- A sentence has been added at the end of this section.

  In the Japanese version, **ShowKeyboard** is not available. Rather, **CorrectWritingEx** allows access to the Data Input Window.

Appendix B Using the 32-Bit Pen Application Programming Interface

32-Bit Functions

- The **KKConvert** function has been added to the list of functions supported by the 32-bit Pen API.