# Microsoft Windows for Pen Computing: Programmer's Reference

# Microsoft® Windows™ for Pen Computing

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software, which includes information contained in any databases, described in this document is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Microsoft Corporation.

Copyright Microsoft Corporation, 1992. All rights reserved.

Microsoft, MS, MS-DOS, and QuickC are registered trademarks and Windows is a trademark of Microsoft Corporation.

Document Number WG28920-0292

# Contents


**Before You Begin** ix

**Chapter 1 Getting Started with Microsoft Windows for Pen Computing** 1

Objective 1
What Is Microsoft Windows for Pen Computing? 1
- Device Drivers 2
- The Recognizer 3
- Microsoft Windows for Pen Computing DLL 3
- The Pen Interface 3

What Is the Microsoft Windows for Pen Computing SDK? 3
How Will MS Windows for Pen Computing Be Delivered? 4
What Are the Basics of the Pen API? 4
- The Handwriting Edit (hedit) Control 4
- The Boxed Edit (bedit) Control 4
- Data Flow (Overview) 5
- ProcessWriting Function 5
- Recognize Function 5
- The RC Structure 5
- The RCRESULT Structure 6

How Should Applications Support the Pen? 6
- Basic Pen Awareness (Pen-Capable Applications) 6
- Extended Pen Awareness (Pen-Enhanced Applications) 8
- Advanced Pen Awareness (Pen-Centric Applications) 9

This Guide 11

**Chapter 2 The Architecture of the Pen Extensions** 13

The Goals 13
Components 14
- The RC Manager 14
- The Pen Driver 14
- The Display Driver 15
- The Pen Message Interpreter 15
- Windows Applications 16
- Pen Applications 16
- Recognizers 16
- Dictionary Modules 17

Data Flow 17

| | | |
|---|---|---|
| | The Pen Driver | 18 |
| | The RC Manager: Normal Mode | 18 |
| | The Application | 20 |
| | The RC Manager: Inking Mode | 21 |
| | The Results | 23 |
| | The Pen Message Interpreter and the Rest of the System | 24 |
| | The Gesture Macro Layer | 24 |
| **Chapter 3** | **The Recognition Process** | **27** |
| | Overview | 27 |
| | RC: The Principal Data Structure | 29 |
| | Using and Modifying Ink | 29 |
| | Ending Recognition: Pen State | 29 |
| | The Application-Recognizer Connection | 30 |
| | Writing Location | 31 |
| | Specifying the Recognizer and the Type of Input Expected | 31 |
| | The Timing of Recognition Results and Significant Events | 33 |
| | Controlling the Recognition Process | 33 |
| | Specifying the User for Recognition | 35 |
| | Dictionary Processing | 36 |
| | Recognition Results: The RCRESULT Structure | 36 |
| | The WM_RCRESULT Message | 37 |
| | Symbols and Symbol Values | 37 |
| | The Symbol Graph | 37 |
| | The Best Guess | 39 |
| | Location and Position of the Input | 39 |
| | Contextual Information | 40 |
| | The Ink | 40 |
| **Chapter 4** | **Managing Ink in Pen Applications** | **41** |
| | The HPENDATA Data Type and Ink | 41 |
| | The Basics | 41 |
| | The Details | 42 |
| | The Ink Functions | 45 |
| | Rendering Pen Data | 45 |
| | Transforming Pen Data | 46 |
| | Pen Data Housekeeping | 47 |
| | Pen Data Input and Output | 48 |
| | Compressing Pen Data | 50 |
| | Display Resolution Compression | 50 |
| | Common Scenarios Using an Ink Object | 51 |

**Chapter 5 A Sample Pen Application 53**

Overview of the PENAPP Application 53
WinMain and Initialization Functions 54
- WinMain 54
- FInitApp 55
- FInitInstance 56
- FLoadRec 58

Data Handling and Display Functions 60
MainWndProc 60
InputWndProc 62
InfoWndProc and RawWndProc 65

**Chapter 6 Using Pen Controls and the ProcessWriting Function 67**

Using the Hedit (Handwriting) Pen Control 67
- Control Messages 68
- WinMain 68
- Initialization Functions 70
- HformWndProc 71
- FCreateForm 72
- Dialog Box Functions 75

Using the Bedit (Boxed Handwriting Edit) Control 75
- Using Bedit Controls in Dialog Boxes 77

Using the ProcessWriting Function 77
- Modifying a Windows Program to Use ProcessWriting 79

**Chapter 7 Replaceable Components: Recognizers and Dictionaries 81**

Recognizers 81
- Converting Input to Usable Data 81
- Returning Results 83
- Training 85
- Symbol Values and Symbol Graphs 85
- The RC Structure 87
- How a Custom Recognizer Interacts with the RC Manager 89
- A Sample Recognizer 91

Dictionaries 96
- The RC Structure and Dictionary Processing 96
- Subfunction Messages Used in a Dictionary DLL 99
- A Sample Dictionary 100

**Chapter 8 Pen API Overview 105**

Pen API Categories 107
Pen Interface Functions 107
Pen Data Functions 113
Custom Recognizer Functions 114
Pen Module Functions 115
Pen Driver Functions 115
Display Driver Functions 116
Dictionary Functions 116

**Chapter 9 Pen API Reference 115**

AddPenEvent 116
AddPointsPenData 117
AtomicVirtualEvent 118
BeginEnumStrokes 119
BoundingRectFromPoints 120
CharacterToSymbol 121
CloseRecognizer 122
CompactPenData 123
ConfigRecognizer 126
CorrectWriting 132
CreatePenData 134
DestroyPenData 136
DictionaryProc 137
DictionarySearch 151
DPtoTP 153
DrawPenData 154
DuplicatePenData 156
EmulatePen 157
EndEnumStrokes 158
EndPenCollection 159
EnumSymbols 160
ExecuteGesture 162
FirstSymbolFromGraph 164
GetGlobalRC 165
GetMessageExtraInfo 167
GetPenAsyncState 168
GetPenDataInfo 169
GetPenDataStroke 170
GetPenHwData 172
GetPenHwEventData 174
GetPointsFromPenData 177
GetSymbolCount 178

GetSymbolMaxLength 165
GetVersionPenWin 166
InitRC 167
InitRecognizer 169
InstallRecognizer 170
IsPenAware 171
IsPenEvent 172
MetricScalePenData 173
OffsetPenData 175
PostVirtualKeyEvent 176
PostVirtualMouseEvent 177
ProcessPenEvent 179
ProcessWriting 180
Recognize 183
RecognizeData 186
RecognizeDataInternal 187
RecognizeInternal 188
RedisplayPenData 190
RegisterPenApp 192
ResizePenData 193
SetGlobalRC 194
SetPenHook 196
SetRecogHook 197
ShowKeyboard 199
SymbolToCharacter 203
TPtoDP 204
TrainContext 205
TrainContextInternal 207
TrainInk 209
TrainInkInternal 211
UninstallRecognizer 212
UpdatePenInfo 213

**Chapter 10 Pen Structures 215**

BOXLAYOUT 216
GUIDE 218
OEMPENINFO 221
PENDATAHEADER 223
PENINFO 225
PENPACKET 228
RC 229
RCRESULT 240
RECTOFS 244

SKBINFO 245
STROKEINFO 246
SYG, SYE, SYC, and SYV 247

**Chapter 11 Pen Messages and Constants 251**

ALC_Values (Alphabet Code) 252
BXD_Values (Boxed Edit Control) 255
HN_ Notification Messages 256
IDC_ Values (Display Cursor) 257
PCM_ Values (Pen Collection Mode Values) 258
PDC_ Values (Pen Device Capabilities) 259
PDK_ Values (Pen Driver State Bits) 260
PDT_ Values (OEM-Specific Data) 261
PDTS_ Values (Data Scaling Values) 262
RCD_ Values (Writing Direction) 263
RCO_Values (Recognition Option) 264
RCOR_Values (Tablet Orientation) 266
RCP_Values (User Preferences) 267
RCRT_Values (Results Type) 268
REC_ Values (Recognition Functions) 270
SYV_ Values (Symbol) 273
WM_GLOBALRCCHANGE Message 276
WM_HEDITCTL Messages 277
WM_RCRESULT Message 282
WM_SKB Message 283

**Appendix A Guide to the Initialization Files 285**

FORMAT of .INI files 285
Modifying the SYSTEM.INI File 286
Modifying the CONTROL.INI File 288
Modifying the PENWIN.INI File 289
Modifying the WIN.INI File 293

**Appendix B Pen Addenda for MS Windows API Functions 295**

GetSystemMetrics 295
SetClipboardData 295
Combo-Box Notification Codes 295
Differences Between Bedit and Edit Controls 295
Installable Pen Device Driver 296

**Index 301**

# Before You Begin

This guide describes the Microsoft® Windows™ graphical environment for Pen Computing Software Development Kit (SDK). The chapters that follow contain information on:

- Installation
- SDK components
- Pen application programming interface (API)

## Organization

This guide is divided into the following chapters and appendixes.

| Chapter | Description |
|---|---|
| Chapter 1, Getting Started with Microsoft Windows for Pen Computing | An overview of pen computing, the Pen SDK, and setup information. |
| Chapter 2, The Architecture of the Pen Extensions | An overview of the architecture of the pen API extensions used to build pen applications. |
| Chapter 3, The Recognition Process | An overview of the data structures and methods used in the recognition process. |
| Chapter 4, Managing Ink in Pen Applications | An overview of the data structures and methods used in the inking process. |
| Chapter 5, A Sample Pen Application | A sample application that uses child windows for handwriting input. |
| Chapter 6, Using Pen Controls and the **ProcessWriting** Function | A sample application that uses the pen controls. This chapter also introduces the **ProcessWriting** function. |
| Chapter 7, Replaceable Components: Recognizers and Dictionaries | Sample recognizer and dictionary dynamic-link libraries (DLLs). |
| Chapter 8, Pen Overview | An overview of the Pen API. |
| Chapter 9, Pen API Reference | Descriptions of the Pen Extensions functions. |
| Chapter 10, Pen Structures | Descriptions of the Pen Extensions structures. |

| Chapter | Description |
| --- | --- |
| Chapter 11, Pen Messages and Constants | Descriptions of the Pen Extensions messages and constants. |
| Appendix A, Guide to the Initialization Files | Settings used in the Windows and Pen Extensions .INI files. |
| Appendix B, Pen Addenda for MS Windows API Functions | Supplemental notes to the Windows 3.1 API documentation. |

# Document Conventions

The following document conventions are used throughout this manual.

| Convention | Description |
| --- | --- |
| **Bold text** | Bold letters indicate a specific term or punctuation mark intended to be used literally: language functions or keywords (such as **InitRC** 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, the following syntax for the **InitRC** function initializes the recognition context for the recognizer:<br><br>**InitRC**(*hwndInput, (LPRC) &rc);* |
| `Monospaced text` | Code examples are displayed in a nonproportional typeface. |
| `if(!RegisterClass(LPWNDCLASS)&wc))`<br>`.`<br>`.`<br>`.`<br>`else` | A vertical ellipsis in a program example indicates that a portion of the program has been omitted. |

| Convention | Description |
|---|---|
| ... | 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. |
| { } | Curly 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. |

# Additional Documentation

This guide assumes a basic familiarity with MS Windows programming. The additional documentation listed in the following table explains the MS Windows Graphical Environment for Pen Computing.

| Title | Contents |
|---|---|
| Microsoft Windows for Pen Computing documentation package | Introduction and tutorial materials for the pen computing system. These materials describe the user interface and Windows applications. |
| Microsoft Windows for Pen Computing online help | The complete source for Windows user documentation. |
| Microsoft Windows Software Development Kit (SDK) documentation, or equivalent documentation | Information about the application programming interface of the Microsoft Windows graphical environment. |
| Microsoft Windows Device Driver Development Kit (DDK) documentation, or equivalent documentation | Describes the application programming interface of the Microsoft Windows device drivers. Required if you are developing pen or tablet drivers. |

# System Requirements

You can develop pen applications with the following software and hardware:

- An IBM personal computer or compatible running Microsoft Windows version 3.1 or later
- A mouse, tablet, or other pointing device supported by the Microsoft Windows for Pen Computing system
- Microsoft Windows version 3.1 SDK
- Microsoft Windows version 3.1 Device-Driver Development Kit (DDK)—necessary only if you build pen, display, tablet, 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 (for example, Borland International Inc.).

# Setting Up

You can run the SETUP.EXE Windows program on Disk 1 of the distribution disks to install this SDK.

---

**Note** Do not install this version of the SDK over any older versions. You must install this SDK in a new directory. (References in this book assume that the SDK has been installed in the default PENSDK directory, however.)

---

Be sure to read the README.TXT file on Disk 1 for late-breaking release notes.

# Getting Started with Microsoft Windows for Pen Computing

This chapter provides background information on Microsoft Windows for Pen Computing. It also discusses pen computing in general, the pen computing application programming interface (API), and interface design considerations appropriate to pen-based applications. This introduction lays the groundwork for your participation in the pen computing revolution with Microsoft Windows for Pen Computing.

## Objective

Since the beginnings in 1988, the goal of the Microsoft Windows for Pen Computing development team has been the creation of a compelling and compatible pen-based operating environment.

A compelling pen-based operating environment enables the creation of applications that extend current graphical user interface (GUI) techniques and interact with the user on a "pen-and-paper" level. This metaphor is exciting because it is familiar to users and less intimidating than the current keyboard and mouse standard. Given the familiarity of the pen and how it is used, the interaction between a user and a computer becomes more natural if a pen is the method of interaction.

A compatible pen-based operating environment interacts with existing Windows applications without modification. Microsoft Windows for Pen Computing offers increased effectiveness to the installed base of Microsoft Windows applications and hardware platforms. By employing the code base and experience of Windows independent software vendors (ISVs), the Pen Extensions widen the acceptance of Microsoft Windows in corporate settings.

MS Windows for Pen Computing provides a migration path for current Windows products and a means by which interested ISVs can create advanced pen-centric applications from the ground up.

## What Is Microsoft Windows for Pen Computing?

Microsoft Windows for Pen Computing, also known as Pen Extensions, is a series of modular extensions to the MS Windows 3.1 operating environment. The Pen Extensions include a set of dynamic-link libraries (DLLs) and drivers that enable pen-based input and handwriting recognition in Microsoft Windows. The components of the Pen Extensions are transparent to the normal Windows 3.1 applications, and yet they are readily available for those applications that seek to leverage their capabilities.

Pen services are available through a new set of APIs, referred to in the following pages as the Pen API. This API is available to every computer running version 3.1 of MS®

Windows—regardless of whether or not that computer is a pen computer or has an attached pen peripheral. Application developers can therefore leverage the Pen API window classes—hedit and bedit—as well as use other pen API services, and feel confident that their programs will run identically on all machines running MS Windows 3.1.

If the pen is present, the Pen API informs applications so they can activate advanced pen-specific features. It also automatically enables pen interaction in the pen control classes. If the pen is not present, the same .EXE will operate without modification—and without pen behaviors, of course—under Windows 3.1.

The Pen Extensions have been designed and built for MS Windows version 3.1, and this version of Windows—or a later version—is required for pen-specific behaviors.

The Pen Extensions can be broken down into four general areas. The following paragraphs summarize them briefly.

## Device Drivers

To use MS Windows for Pen Computing, you need to install a pen driver, and you generally need to use a modified standard display driver.

### Pen Drivers

A pen driver is an installable device driver. Its primary role is to get data from the digitizing device into the Windows system. The data from a digitizing device consists minimally of (x,y) coordinates indicating pen position; it may also contain pressure or angle information. All information reported by a pen driver is available to applications should they decide to use it.

Pen drivers are distinct from mouse drivers in three important ways. First, they report data at much higher sampling rates and at much higher resolution than a mouse does. This is required to support handwriting recognition. Second, they may also report pressure, angle, rotation, or other pen state information. Finally, they employ a private pen computing interface to manage the high data rate and density, thereby avoiding a flood of useless information.

### Display Drivers

A quality pen user interface requires the ability to ink (that is, to draw lines to the current pen location to give the illusion that the user is drawing on the screen. To improve performance and reduce code duplication, Microsoft has implemented inking in the RC Manager.

To support the inking process, a communication is established between a Windows display driver and the pen interface in such a way that ink can be drawn at interrupt time. Thus, in much the same fashion as the mouse cursor follows the mouse immediately, ink follows the pen immediately by a close interaction between MS Windows for Pen Computing and the Windows display driver at interrupt time.

## The Recognizer

The portion of the system that actually turns streams of (x,y) points into recognized characters is a DLL referred to as the *recognizer.* The Microsoft recognizer recognizes neatly hand-printed characters from the ANSI character set. However, the recognizer component can be replaced by third-party recognition systems that may offer improved recognition rates, different symbol sets, and other qualities not yet available in the Microsoft recognizer. Several other manufacturers have already begun to develop recognizers for pen computing systems.

In addition to the standard characters, a recognizer might also recognize circles, squares, triangles, Kanji symbols, Gregg shorthand, mathematical symbols, CAD/CAM symbols, and other symbols or characters. MS Windows for Pen Computing SDK includes a special recognizer (SHAPEREC.DLL) that recognizes circles, ellipses, squares, rectangles, and lines.

## Microsoft Windows for Pen Computing DLL

PENWIN.DLL is the manager of all pen-specific components in the Windows 3.1 system other than those handled by the recognizer. PENWIN.DLL is the implementation point for the majority of the Pen APIs and acts as an intermediary between the pen driver, the display driver, the recognizer, and pen-aware Windows applications.

PENWIN.DLL is included with the retail versions of MS Windows 3.1. At runtime, PENWIN.DLL determines whether or not there is a pen attached to the system and takes appropriate action. It is the presence of this DLL in all Windows 3.1 systems that ensures that developers can always leverage hedit, bedit, and any of the other Pen APIs.

## The Pen Interface

There are a number of new interface components in MS Windows for Pen Computing that ease or otherwise assist pen interaction, accessed primarily through a small floating toolbar called the Pen Palette. These new interface components include the Trainer, the Gesture Editor, the On-Screen Keyboard, and the Writing Window. Also included are four new Control Panel extensions that provide a means to identify specific users to the system, set user-configurable options, determine screen orientation, and calibrate the stylus device. Discussed in detail later in this guide, these elements provide a primary interface with which users manage and modify their pen computing environment.

# What Is the Microsoft Windows for Pen Computing SDK?

The Microsoft Windows for Pen Computing SDK (Pen SDK) is an extension to the MS Windows version 3.1 SDK. The Pen SDK contains libraries, INCLUDE files, and numerous sample sources. The Pen SDK enables a Windows ISV to develop pen-based applications and custom recognizers.

The MS Windows version 3.1 SDK is also required for the development of pen-aware applications and recognizers. The Pen SDK assumes that the INCLUDE files, libraries, and tools in the Windows 3.1 SDK are present.

Any development tool that can compile Windows APIs in C code can do the same with the Pen API and libraries as long as the development tool calls the Windows 3.1 functions directly and links to the standard API libraries included with MS Windows 3.1. These capabilities indicate that the tool is sufficient for the same functions with the MS Windows for Pen Computing SDK elements.

# How Will MS Windows for Pen Computing Be Delivered?

MS Windows for Pen Computing is an original equipment manufacturer (OEM) product. This means that both developers and end users purchase the product only when they buy hardware that uses its capabilities. The hardware can be a portable pen-based computer, a notebook machine with a screen that can be removed and carried around as a pen-based computer, or a digitizing tablet used on the desktop.

It is important to note that the Pen SDK mentioned previously does not include the MS Windows for Pen Computing runtime components. You can test your pen applications under Windows 3.1 without a pen; however, you cannot test them fully until you acquire pen hardware.

# What Are the Basics of the Pen API?

This section assumes that you are familiar with standard Microsoft Windows programming conventions. It is provided as an introduction to the detailed discussion that follows in the rest of this guide.

There are seven primary elements for the development of pen-specific behaviors in Windows applications: the handwriting, or hedit, control; the boxed edit, or bedit, control; the data flow; the **ProcessWriting** function; the **Recognize** function; the **RC** structure; and the **RCResult** structure.

## The Handwriting Edit (hedit) Control

The hedit control is a replacement for the edit control. The hedit is a version of the normal Windows edit control that accepts handwritten input and allows for the configuration and control of the recognition process through a messaging interface.

## The Boxed Edit (bedit) Control

The bedit control is an entirely new Windows control, implementing boxed input. The following is a boxed edit control showing the cells in the letter guides. Sometimes this control is referred to as a *comb*.

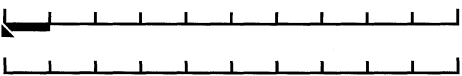


Figure 1.1. Boxed edit control

The message set supported by the bedit control is essentially a superset of that supported by the standard edit control, with additions provided for modifying of the recognition process and rendering the letter guides. This control was designed from the ground up for the pen, but it also supports a keyboard interface for compatibility with machines with attachable keyboards.

## Data Flow (Overview)

The data flow in the Pen Extensions is similar to that of a regular Windows program. The pen driver reports events to Windows as if they were mouse events. Applications treat the pen as a mouse the majority of the time.

The action begins when the user puts the pen down (mapped to WM_LBUTTONDOWN) in an area—or over a window—that the application has determined to be a writing area. When an application determines that this has occurred, it calls the **ProcessWriting** or the **Recognize** function and awaits the results. Both functions activate inking, process the recognition of ink, and package up the results to be returned to applications.

For a more detailed discussion of the data flow, see Chapter 2, "The Architecture of the Pen Extensions."

## ProcessWriting Function

An application uses the **ProcessWriting** function to request that all of the basic recognition parameters be used and that the results be returned as WM_CHARs. This is the quickest way to add handwriting capabilities to an existing Windows application, because the application is insulated from the complexities of managing recognition results. An application receives a WM_LBUTTONDOWN message and responds by calling **ProcessWriting**. WM_CHARs are received, **ProcessWriting** returns, and processing resumes.

## Recognize Function

If the **ProcessWriting** function is the easiest way to add handwriting capability to an existing Windows application, the **Recognize** function can be considered the most flexible. The caller has complete control over the recognition process. The results are returned through a new message called WM_RCRESULT before **Recognize** returns. An application that needs to perform exacting control over the recognition process or implement some of the more advanced features might use this function instead of **ProcessWriting**.

## The RC Structure

The **RC** structure, the primary controlling data structure of the Pen API, is used by the application to moderate the recognition process when **Recognize** or **ProcessWriting** is called. The **RC** controls such parameters as ink width and color, the recognizer to be used, the timing of the results, which user actions terminate inking, the type of input expected (that is, numeric or alphabetic), and other parameters.

### The RCRESULT Structure

This data structure is used to pass recognition results back to applications that have called **Recognize** or **ProcessWriting**. The WM_RCRESULT message contains a pointer to an **RCRESULT** structure. **RCRESULT** contains the recognizer's best guess as to what the user entered, the possible alternatives, and the actual ink data entered by the user.

## How Should Applications Support the Pen?

The following three-tiered breakdown of pen enhancements can be viewed both as a suggestion list for new pen-aware applications and as guidelines for modifying existing Windows applications. By following these guidelines you will be able to see where your application must be modified if it is to be run on pen-based systems, where it might be augmented to support the pen specifically, and where value can be added through the incorporation of advanced pen-specific behaviors.

*Important* There is no substitute for testing your application on an integrated display-digitizer (if not an actual pen-based computer) to understand the strengths and limitations of the pen. Specifically, try your application on such a device to determine how well it works with a pen. This type of usability testing can help you see which pen interfaces are more appropriate than others.

### Basic Pen Awareness (Pen-Capable Applications)

Basic pen awareness is the groundwork necessary to provide a minimally functional pen-based interface and a base for further pen-aware behaviors. The effort necessary to achieve basic pen awareness is minimal; the resulting application is fully compatible with desktop-oriented applications.

#### Registering Your Application with RegisterPenApp

Calling the **RegisterPenApp** function will result in the replacement of all edit windows created in your application by hedit Windows. Note that this includes the edit field portion of combo boxes. This is the first step to being pen-aware. However, you should make sure that no subclassing of standard edit controls is broken when they are replaced with hedit controls.

#### Handwriting Recognition Is Difficult

Handwriting recognition presents a problem that is very difficult to solve to a user's satisfaction. The first thing to do to an application is to limit the amount of handwriting required to the greatest extent possible. For example, if you can provide a combo box with a list of acceptable inputs, provide it. Picking something from a list will always result in 100 percent recognition.

Similarly, buttons always result in 100 percent recognition. You will find that using buttons—as in toolbars—and selection lists will improve the usability of an application whether used with or without the pen.

## Writing Areas

Provide writing areas within your application as appropriate, to allow for text and gesture entry. The **ProcessWriting** function is provided for this purpose. It allows support for the standard editing gestures and text entry with minimum effort on the part of the programmer. The PENPAD sample in the PENSDK\SAMPLES\PENPAD directory of the Pen SDK uses this function.

## Power Management

Pen-aware Windows applications—and, increasingly, all MS Windows 3.1 applications—will run on portable systems with limited battery life. New power-management facilities in Windows 3.1 provide for the detection of idle time in Windows applications and institute special power-saving mechanisms when the system is idle. However, applications that spin in a **PeekMessage** loop as their main message loop are never idle. Applications that do not call **GetMessage** or **WaitMessage** will prematurely exhaust the battery of any portable machine, because they short-circuit the Windows idle-detection mechanism.

## Display Considerations

Pen-aware Windows applications need to run acceptably on monochrome monitors, because a number of early pen-based machines will have them. Application designers should consider this when designing bitmaps and color schemes.

When designing applications, you should also bear in mind that the configuration of the desktop is likely to change from execution to execution because of the rotation of the screen from portrait to landscape mode. Therefore, Windows programs should determine display dimensions dynamically with each execution. You should take special care when designing toolbars and dialog boxes, because it is deceptively easy to design both items in such a way that all or part of them is off the screen when the display is in portrait mode.

## Avoiding Keyboard-Only Behaviors

Beware of creating commands in your applications that are dependent on keyboard shortcuts or key modifiers—for example, a zoom feature that can be restored to normal view only by pressing the ESC key. If the user doesn't have a keyboard, there will be no way to restore normal view. A button or gesture could be used instead of a key. This is yet another example for which a toolbar would provide an excellent and immediate interface beneficial to both the pen and the mouse user.

## Understanding Portable Platforms

Because Microsoft Windows will increasingly find its way onto highly portable platforms, the application vendor should strive to comply with the size and storage limitations inherent in such platforms.

There are a number of strategies for ensuring this. One is to use compressed file formats to store application data. Another is to break up an application—or its data dependencies—into a number of pieces (DLLs) that can be "left behind" if only a subset of an application's functionality is required. In general, where vast amounts of fixed storage could once be counted on, a pen-based machine may not have as much storage available.

### Other Considerations

The "text goes where you write it" rule should be implemented whenever reasonable. The user should not have to tap in a field or window before writing is accepted by it. This is one of the features of the hedit class. Any writing areas in your application should do the same.

The pen is very good at indicating general positions—at pointing to regions, selecting menu items, hitting buttons, and so on. It is not good at indicating specific locations on the screen. To test this, you might try tapping on a single display pixel with a pen. This should influence your decisions about button size, the size of the handles you put on objects, font size, and so on.

## Extended Pen Awareness (Pen-Enhanced Applications)

*Extended pen awareness* is the quality describing applications that provide functionalities specific to the pen. There are a number of elements that help make pen computing easy, because they are based on modes of interaction that are already familiar to the user. These are discussed in the following paragraphs.

### Boxed Edit (bedit) Controls

Boxed input is an extremely powerful way to get handwritten input from the user and should be used in your pen-aware applications whenever possible. The advantages of bedit controls are numerous:

- The recognizer has excellent segmentation and baseline information.
- Users write more neatly when constrained with boxes.
- The boxed control interface is especially natural and appropriate for pen input.

Note that the bedit control is less than optimal when the amount of user input cannot be predicted or restrained. It functions best when the input is known to be of a certain length and type—for example, a social security number, a phone number, or a first name.

### Ink Field: Retaining Ink as Data

The ability for a user to enter ink and store it as ink is extremely powerful. The hedit and bedit classes support this already, but it requires little extra effort to add the inking capabilities to an existing window class. By storing ink in analog format, the problems associated with handwriting recognition can be avoided, and yet the user is able to understand and manage the information contained in the ink. In essence, this is the electronic equivalent of taking notes on scraps of paper.

If ink is entered as graphical data, the ability to recognize that data later is crucial. In MS Windows for Pen Computing, delayed recognition is implemented in hedit and bedit controls. The **RecognizeData** function in the Pen API enables an application to implement delayed recognition on its own stored ink.

Application developers should not ignore the importance of providing inking capabilities in applications. It is easy to implement inking behavior with the Pen API. The power and usability this feature adds to your application is tremendous.

### Context

A pen-aware application can provide contextual information that is applied at the beginning of the recognition process, and contextual information that is applied at the end of the process. By providing context, applications improve their recognition rate substantially.

For example, before recognition begins, it is useful for the recognizer to know what type of data to expect. If a field is numeric, the recognizer should be programmed to expect only numeric data. If the field is alphabetic, the recognizer should be programmed to expect only alphabetic data. Other types of contextual information include gestures only, Kanji symbols only, and normal alphanumeric characters. The hedit and bedit controls support a message interface through which contextual information can be provided, and the **RC** structure provides the same information when **ProcessWriting** and **Recognize** are used.

In addition to providing preprocessing clues to the recognizer, MS Windows for Pen Computing provides a dictionary path to check recognized pen input against a set of words (or multiple sets of words) to aid in the recognition process. A dictionary is a DLL that communicates with PENWIN.DLL to help determine which of the recognizer's guesses is the best guess. This facility can be folded into your applications where appropriate. For example, lists of states, terms specific to your application, user names, keywords, or any other anticipated input can be provided as a check against the recognition process.

In this way, even very poor handwritten input can be interpreted correctly by comparing it to the set of expected inputs. A dictionary for the English language is included with MS Windows for Pen Computing to provide this capability for normal English input.

### Good Graphical User Interfaces II

Besides the suggestions offered in "Handwriting Recognition is Difficult," earlier in this chapter, there are further basic GUI elements that you can focus on to ensure that good pen interaction is possible with your product. These include object linking and embedding (OLE), direct manipulation, the use of toolbars, and a generally uncluttered user interface.

## Advanced Pen Awareness (Pen-Centric Applications)

Pen-centric functionalities are directed solely at a pen-based platform. They are usually too complex to manage with only a keyboard and a mouse. A personal information manager (PIM) application targeted at portable pen-based machines would fall into this category. Pen-centric behaviors and functionalities are many; the following are a few general ideas.

## Text Goes Where You Write It

There are several terms used to describe this behavior; *smart targeting* is one of the best. The idea is that if a user writes text at a particular location, the application should understand where that text was really meant to go and make sure it gets there. This functionality might boil down to something as simple as electronic paper—that is, text is recognized where it is written and placed in an object for later management by the user.

## Annotation

An annotation layer, in contrast to an inking field, tends to be all-encompassing and not constrained to an individual field or location. The ability for a user to scribble all over an application window, print the annotations, select and modify them, hide them, and so on, is a complex and powerful pen-centric behavior. An example of this type of functionality would be the capability to annotate a word-processing document and pass your handwritten edits along with the text of the document back to the author for revision.

## Special Recognition and Shape Recognition

Another powerful pen-centric functionality stems from the ability of special recognizers to convert glyphs to application- and context-specific input. The shape recognizer included with MS Windows for Pen Computing is one such recognizer.

A drawing package that can *snap,* or instantly reconfigure, a rough circle or square to one that is true is especially valuable. For example, a CAD/CAM application can have a special recognizer designed to recognize the symbols specific to the industry in which it is employed.

## Pen and Paper

The pen-and-paper metaphor refers to a shift of responsibility from the user to the application; the application makes decisions and proceeds based on an understanding that the application work area resembles paper.

Functionalities of this class include context-sensitive pen input. Consider the example of a scribble entered in a drawing region of an application. If a shape recognizer is unable to determine what the scribble is, it retains it as a scribble; if it recognizes the scribble, it snaps it to a circle, square, triangle, or line, as appropriate. If the same glyph is entered in a writing region of an application, it is translated to a letter.

Another example of pen-and-paper behaviors is known as *math paper.* If a region of your application is designated as math-aware, it understands how to perform calculations as they are written by a user. For example, if your application understands that an equal sign is a part of the handwritten input, then the input to the left of the equal sign should be evaluated as an equation, and the answer must be provided as part of the recognized input.

Another common attribute of this class of application is the ability to bypass the cursor. With a pen, the user indicates the location of the pointing device by placing the pen tip on top of it, thereby reducing the need for the cursor. In practice, you may find that this is

only partly true, but when visual feedback is moved from the cursor to a change in the object or area indicated with the pen, it is both powerful and easy to grasp for the average user. For example, the handles for an object appear only as you move the pen over the edge of the object.

# This Guide

The remainder of this programmer's guide provides in-depth converage of many of the topics touched upon in the preceding pages. Included are a detailed background on the architecture of MS Windows for Pen Computing and information on the use of the **RC** and **RCRESULT** structures. Information about adding ink support to your application is also included. Combined with the MS Windows API reference, this guide will serve as a roadmap to the pen API and pen computing functionalities in your applications.

# The Architecture of the Pen Extensions

This chapter discusses the architecture and components of Microsoft Windows for Pen Computing and how these elements interact with the Windows version 3.1 system. The reader is assumed to be familiar with basic Windows architecture, programming techniques, and naming conventions.

## The Goals

The goal of the Pen Extensions is to give independent software vendors (ISVs) the ability to create pen applications that run on top of Microsoft Windows. Powerful pen applications require something more than simple mouse and keyboard emulation. A flexible application programming interface (API) must allow ISVs complete freedom when designing their pen interfaces. The API must enable "pen-and-paper" interaction with the user in a natural and intuitive manner.

These objectives require several behaviors:

- The pen must behave in a manner the user expects. For example, it must leave ink as it moves across the surface of the tablet in the same way that a pen leaves ink on paper. Also, the pen interface should be natural and easy to use.
- The user must be able to indicate command and position in a single pen action, called a *gesture.* The use of gestures is unique to the combination of a pen and a computer, and it gives users certain advantages over normal pen-and-paper interaction.
- Applications must have access to all recognition results and their alternatives; in addition, they must be able to process the ink as the user enters it. This flexibility enables an application to handle user input in the manner appropriate to the task—leaving the information as ink on the page, interpreting it as text characters or shapes, or providing a list of alternatives to the user so that the correct result can be obtained.
- Applications should be able to create and make use of customized recognizers for mathematical symbols, Gregg shorthand, and other specialized handwriting recognition purposes. The Pen API provides a mechanism for passing a single piece of user input to several recognizers, so that if one cannot determine the correct recognition results, another might.
- The installed base of Windows applications should have some means of interacting with the pen. This should certainly take the form of simple mouse emulation and a writing window from which recognized results can be sent to applications. In addition, however, this behavior should work in all writing areas in the system—at least with a minimum of pen functionality (without true pen-and-paper capabilities)—regardless of whether or not the application has been specifically modified to support the pen.

# Components

The following diagram illustrates the relationships of the various components that constitute MS Windows for Pen Computing and their relationships to Microsoft Windows and Windows applications.

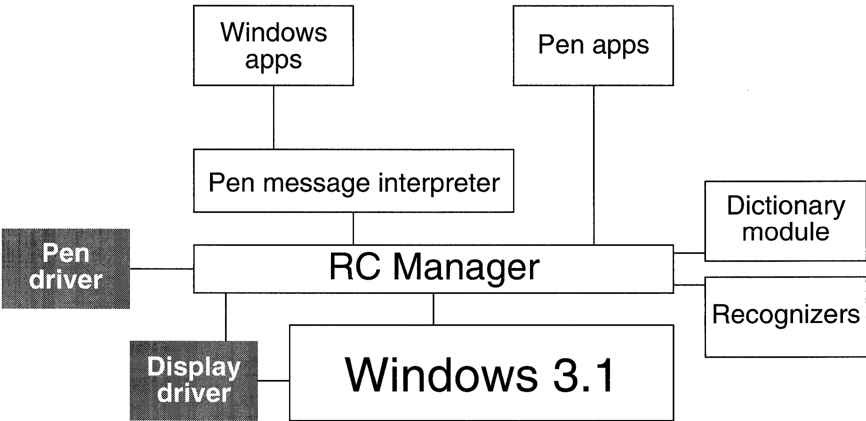



Figure 2.1. The Pen Extensions

## The RC Manager

The RC (Recognition Context) Manager is the heart of the Pen Extensions. If you are familiar with display contexts (DCs), understanding an RC should be straightforward. In the same way that a DC contains all of the information necessary to send graphical output to a device, an RC contains all of the information necessary to carry out pen interaction and handwriting recognition.

The RC Manager moderates the recognition process. It manages the interactions with all of the pen extension components necessary to support the pen and pen behaviors. It records points from the pen driver and passes them on to Windows, serves as the implementation point for the new Pen APIs, integrates the work of the recognizer and dictionaries, and packages up results for applications.

The RC Manager is implemented in PENWIN.DLL, which is analogous to USER.EXE in Windows. Throughout this document, "PENWIN.DLL" and "RC Manager" will be used interchangeably when referring to the Pen Extensions.

## The Pen Driver

This is the component of Windows for Pens that interacts with the pen hardware and passes the information along to the rest of the Windows system by way of PENWIN.DLL.

Two files are associated with the pen driver: an installable Windows device driver that uses the new installable driver interface of Windows version 3.1, and a virtual device driver that handles interaction with the hardware when Windows is running in enhanced mode.

The fact that the pen driver's input may at times be needed for 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 100 times per second. This rate ensures that the true path of the pen is reported accurately enough to support the efforts of vector-based recognizers, and it makes the ink dropped by the pen appear smooth and natural for normal writing speeds.
- The pen driver must be able to report pen positions with a resolution of 200 points per inch. This degree of resolution ensures sufficient granularity in ink coordinates to make accurate judgments about the path of the pen over the digitizing surface. In other words, it ensures that the positions reported are fine enough for a recognizer to derive useful information from them.
- Regardless of the actual resolution of the device, the pen driver must report the pen position in coordinates of 0.001 inch for the RC Manager, the recognizers, and applications to manage the ink in a known and standard scale.

## The Display Driver

As with Windows, the display driver is responsible for interacting with the display hardware and the graphical device interface (GDI) module of Microsoft Windows. For the Pen Extensions, a normal Windows display driver must be enhanced to support inking. Inking support takes the form of two entry points within the display driver and the ability to be called at interrupt time by the RC Manager to perform inking. The entry points are the following.

| Function call | Purpose |
|---|---|
| **InkReady** function | Called by the RC Manager. This call instructs the display driver that, when ready, it should call back the RC Manager to draw some ink. |
| **GetLPDevice** function | Returns a value necessary for the RC Manager's ink-drawing function. |

In addition, the display driver should provide a cursor in the shape of a pen.

In all other respects, the display driver requirements and responsibilities are the same as those for a standard Windows 3.1 driver. For more details on pen-enabled display drivers, see the Windows 3.1 DDK.

## The Pen Message Interpreter

The Pen message interpreter is the component of MS Windows for Pen Computing that interacts with Windows applications that have not been modified to take advantage of the Pen Extensions. This component is responsible for interpreting gestures, handwritten input, and pen events into the corresponding mouse and keyboard events for use by applications that do not do so for themselves.

The most significant feature of the Pen message interpreter is its ability to enable handwriting in all Windows applications. This is accomplished by detection of the I-Beam

cursor. It can do this regardless of whether or not the application uses the Pen Extensions API.

Since most applications use the system I-beam cursor in writing areas, the Pen Extensions can detect this and allow handwritten input of gestures and characters in those writing areas. Once the user begins writing, the Pen message interpreter intercepts the normal pen-to-application data flow and acts as an intermediary between the old application and the Pen Extensions.

As pen input is received by the Pen message interpreter, it is translated into the appropriate mouse and keyboard messages, which are passed on to the application. The application has no knowledge of the pen—or that pen input has occurred.

There are limitations in the compatibility provided by the Pen Message Interpreter. First, it cannot handle nonstandard Windows applications (for example, those without a standard I-beam cursor). Second, applications developed for MS Windows prior to version 3.1 were not designed with the pen in mind, and therefore they may not work optimally with the pen; for example, an edit field may be too small to write in. Finally, no contextual information from the application is available to the Pen message interpreter. This adversely affects recognition rates.

These basic limitations mean that applications must call the Pen Extensions API directly to implement the highly positional, context-dependent behaviors that define good pen applications.

## Windows Applications

The Windows Apps box in Figure 2.1 represents unmodified Windows applications executing in standard or enhanced mode.

## Pen Applications

The Pen Applications box represents new applications that call the Pen Extensions API directly. They bypass the Pen message interpreter to provide more direct, intuitive, and advanced pen interface functionalities. Often these take the form of positionality behaviors that the Pen message interpreter cannot interpret.

For example, a circle drawn by the user might be the letter "O" in a writing area; however, if it's drawn in a drawing area, it might be a shape that should be snapped to an exact circle; if drawn in a scratch area, it might be intended to remain as ink; if drawn over an object, it might select the object. The application's ability to process the user input as it is intended is possible because the application calls the API directly and can make the correct interpretation based on the location and context of a user's input.

## Recognizers

Any recognizer is a dynamic-link library (DLL) that communicates with the RC Manager through a defined protocol. As the name suggests, the recognizer is responsible for interpreting pen input into recognized symbols. The symbols recognized might be ANSI characters, mathematical symbols, the symbols associated with electrical diagrams, or any other set of symbols that someone decides should be directly recognizable.

The recognizer is completely replaceable and modular. This means that an application can send one sample of user input (ink) to a number of different recognizers, and then examine the results.

### The Microsoft Recognizer

The MS recognizer is a vector-based recognizer. It analyzes the points entered by the user not as an image, but as a succession of positions; when broken down correctly, these positions yield a set of features against which comparisons can be made.

The order in which the user enters points is very important. This can be a stumbling block for users who cannot decide why a recognizer might not be able to distinguish between two very similar characters. The problem can be solved to some extent by multiple character prototypes.

The Microsoft recognizer:

- supports the ANSI character set
- supports delayed strokes—for example, crossing a "t" after completing the rest of a word.
- supports the standard Pen Extensions gestures and the circled letters of the alphabet
- is trainable—that is, it can learn the peculiarities of a person's handwriting to achieve higher recognition rates

## Dictionary Modules

The dictionary module provides a means for checking the results of any recognition against a set of words—or, more generally, a set of acceptable results.

A dictionary is a DLL that communicates with the RC Manager. After the RC Manager receives a result from a recognizer, it passes the result on to the Dictionary, which then has the opportunity to correct that result. Note that a dictionary might be a common English language dictionary, a dictionary of medical terms, a set of proper names, or the like. There can be multiple dictionary DLLs in the dictionary module, and the RC Manager calls each of the dictionaries in turn. The application controls the number of dictionaries called and their calling order.

A good way to think about the dictionary path is as a general means to perform postprocessing on a recognition result.

# Data Flow

Figure 2.2 illustrates the flow of pen input through the system. The following sections discuss how the pen interacts with Windows, unmodified Windows applications, and applications designed specifically for the pen.

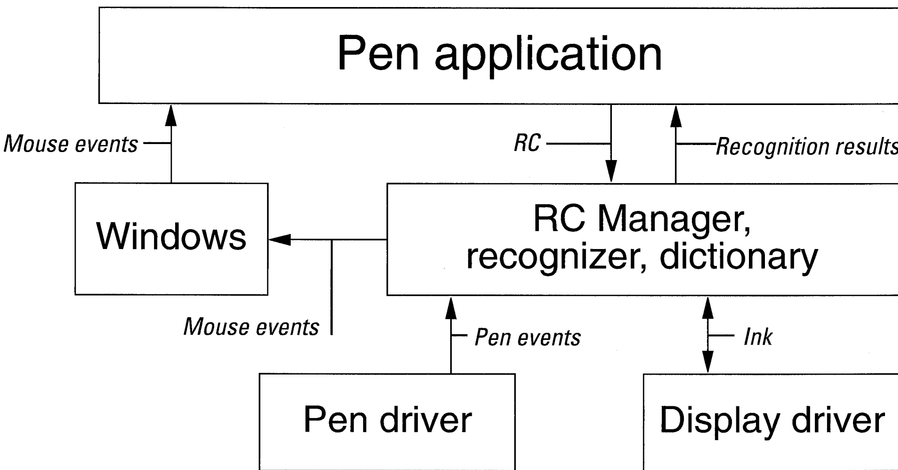



Figure 2.2. Pen Extensions data flow: overview

## The Pen Driver

A good place to start is with the stylus device, or pen driver. Just as the Windows system is driven by keyboard and mouse input, the Pen Extensions are driven by pen input.

If no pen behavior has been requested by an application, the pen behaves as a mouse. It reports the pen status at least 100 times per second to the RC Manager by two function calls in the RC Manager, **AddPenEvent** and **ProcessPenEvent**. Note that the pen driver might be reporting information other than simple x and y position. For example, applications may be requesting the pressure with which the user entered the point, the angle of the pen when it was written, the rotation of the pen, and the distance of the pen from the tablet surface.

The **AddPenEvent** and **ProcessPenEvent** functions pass all pen input to PENWIN.DLL (the RC Manager), which in turn processes the input. Just how that input is processed by the RC Manager depends on the recognition mode, as is discussed in the following section.

## The RC Manager: Normal Mode

There are two possible modes for pen input: normal mode and inking mode.

In normal mode, the pen acts as a pointing device; that is, it is used for pointing and clicking, dragging menus, making selections, and other mouse-like functions. The Pen Extensions need to do very little except act as a message pump from the pen to the Windows system, converting pen events into their Windows mouse message counterparts.

In inking mode, the pen acts as a pen—dropping ink, passing data off to a recognizer, and performing other pen functions. In this mode, Windows is not involved in the process at all. The RC Manager, pen driver, recognizer, and display driver enter a closed universe where they and they alone process pen input. The pen driver reports it, the recognizer recognizes it, the display driver draws it, and the RC Manager processes the interactions necessary to do all of the foregoing.

The transition between modes takes place when an application—or, more precisely, a window—requests that pen input begin. At all other times (and by default), the pen is in normal mode. From the user's point of view, the pen behaves correctly—the pen inks, points, and clicks where it should.

The following paragraphs discuss how the RC Manager processes data when the pen is in normal mode. Inking mode is discussed later in this chapter.

## Buffering Pen Data

The RC Manager buffers all data it passes on to Windows. This is necessary for a number of reasons.

First, the digitizing device operates at a very high resolution with respect to the screen. Because a recognizer requires this resolution to do a reasonable job of recognizing handwritten input, you cannot rely on WM_MOUSEMOVE resolution information for recognition. Therefore, the high-resolution (x,y) information you do not pass along to Windows must be stored so that it is available later when needed for recognition.

Second, the digitizing device may also report information other than simple (x,y) information such as pressure. Windows has no facility for storing this information.

Third, pen events are interrupt-level events with very high report rates. Because all input in Windows is processed synchronously, it is possible that an application will take a very long time to process some message while input continues to stream in. The Windows buffer is not sufficiently large to store all of this information. Therefore, the RC Manager must store it locally to ensure that no data is lost.

Finally, Windows coalesces mouse moves. Because an application is generally interested only in the final location of the mouse, Windows does not report mouse moves associated with old mouse locations. This granularity of measurement is not sufficient for accurate recognition where every location of the pen is important.

## The Message Pump

While the pen is in normal mode, the RC Manager must simply interpret the pen events into the appropriate mouse events and pass the information along to Windows. The RC Manager and pen driver combination creates what amounts to a mouse driver.

The mappings are very straightforward. If the pen touches the surface of the digitizer, the RC Manager sends a WM_LBUTTONDOWN message to Windows. If the pen moves across the surface of the digitizer, WM_MOUSEMOVE messages are generated. This may happen whether the pen is in contact with the surface or not; some devices support the detection of the pen even when it is not in contact with the digitizer. If the pen leaves the surface of the digitizer, a WM_LBUTTONUP message is passed on to Windows.

After messages have been reported to Windows, the generation of double taps, the coalescing of mouse messages, the synchronous handling of input, and other processes are all handled by Windows.

## The Application

The WM_LBUTTONDOWN message notifies a Windows application that inking or recognition should begin. When the pen touches the tablet surface, the result is an WM_LBUTTONDOWN message; it is up to the application to understand that this message has occurred over an area that should be an inking area.

For example, menus used with a pen behave as they do with a mouse. This is because the user expects the pen to work this way with menus. However, in a text area, an application will want to begin inking, because this is what the user expects. When an application receives a WM_LBUTTONDOWN message, it must call back into the RC Manager and instruct the Pen Extensions to begin inking, recognizing the ink, and managing all pen interaction until the recognition event is over.

The API used by an application to accomplish all of this is **Recognize**.

### GetMessageExtraInfo Function: Digitizing Events, Mouse Messages, and Time Travel

You may have noticed a problem with the scheme suggested in the preceding paragraphs—mapping pen events to WM_LBUTTONDOWN and waiting for the application to call back into the RC Manager to begin recognition. It is not possible to determine which (x,y) event from the digitizing device (stored in the private RC Manager buffer) maps to the mouse event being processed by the application.

While Windows input is processed synchronously, pen events are streaming in at 100 points per second. Regardless of what Windows applications are doing—for example, a spreadsheet may be performing a recalculation—the interrupts from the digitizing device continue to stream in, and new (x,y) positions for the pen continue to be recorded in the private RC Manager buffer.

By the time the application calls **Recognize** function, it is not possible to determine which of the buffered events are associated with the mouse message being processed by the application. The solution to this problem requires backtracking and associating the WM_LBUTTONDOWN message being processed by the application with an event buffered by the RC Manager. This facility is provided by **GetMessageExtraInfo** function, a new Windows 3.1 API.

The pen extensions are reporting more than just (x,y) information when they call the Windows **Mouse_Event** function entry point to report an (x,y) position. They are also reporting a 32-bit number that is stored by Windows with the message. This 32-bit number is a pointer into the RC Manager buffer that links the Windows message to a specific digitizer event. Because this extra message information is retained by Windows for the life of the message, the information is available to applications regardless of how long it takes an application to get to it.

Using **GetMessageExtraInfo** function, an application can access this pointer in the RC Manager buffer while processing the WM_LBUTTONDOWN message and pass it into the RC Manager as one of the arguments of **Recognize** function. With this pointer, the RC Manager knows where the buffered events from the digitizer become significant, and it can then safely understand which data to begin sending to the recognizer.

## The RC Manager: Inking Mode

After the **Recognize** function call, the pen device enters pen mode and the recognizer begins to get data and perform recognition. While the RC Manager is in pen mode, the data flow through the Pen Extensions takes the path illustrated in Figure 2.3.

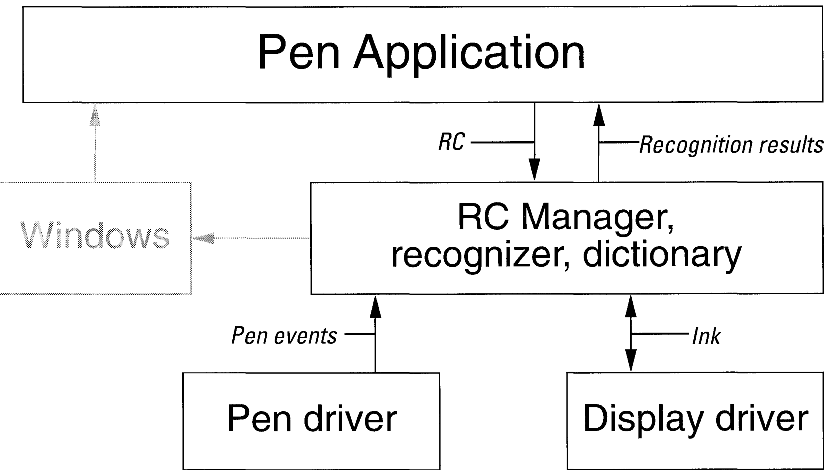



Figure 2.3. Pen Extensions data flow: ink mode

The important point of Figure 2.3 is that Windows is out of the loop. The Pen Extensions interact without the rest of Windows.

Once an application has called **Recognize**, there are three major functions to be carried out before the results can be packaged up and returned to applications. They are inking, recognition, and dictionary processing.

### Inking

It is important that ink be left by the pen in the same way a pen leaves ink on paper. This implies a timely display of bits, as well as accurate positionality information that associates a point on a digitizing device with a point on the display.

Note that the digitizing device and the display are disjoint devices. It is up to the drivers to calibrate their relationship so that pen and ink positions coincide. The Pen Extensions provide a calibration interface for doing just this—at least for simple (x,y) drift. Rotation and compression of the digitizing matrix are not addressed in the tablet calibration item (Calibrate) in the Control Panel window.

The inking procedure is simple. After the pen driver calls **AddPenEvent** and **ProcessPenEvent** to record new data, the RC Manager calls into the display driver informing it that there is ink to be drawn. Then the display driver does one of two things. If it is busy—for example, it is in the middle of a large bitblt—then it must wait to draw the ink until it has finished the other operation. Once it has finished, or if it was not busy to begin with, it calls back into the RC Manager, which in turn draws the ink. The RC

Manager calls a display driver entry point directly to draw the ink. In this respect, the RC Manager behaves exactly as the Windows GDI.

The reason the RC Manager calls the display driver directly is that ink is being drawn at interrupt time. Because GDI is not re-entrant, you cannot rely on it for a timely display of ink on the screen. It was necessary to design a direct interface between the RC Manager and the display driver to avoid calling other portions of Windows. The availability of a known set of display driver functions makes this safe as long as the display driver controls the timing of the whole procedure. The display driver makes sure it is a safe time for the RC Manager to draw the ink by initiating the process with a callback into the RC Manager.

## Recognition

While the display driver and the RC Manager carry out inking, the recognizer is recognizing the points associated with the ink being drawn to the screen.

The RC Manager calls the **RecognizeInternal** function exported by all recognizers. The recognizer then enters a loop, querying data in the RC Manager buffer. The recognizer then interprets the data as character (or other) symbols. Logically, the loop looks something like the following:

```
WHILE ( GetMoreXYData() != TERMINATION_EVENT ) {
    IF ( ThereWerePoints)
        Process()
}
SendResultsToDictionariesAndApplications()
return
```

A number of different events can terminate recognition:

- The pen leaving a bounding rectangle provided by the calling application
- The pen entering an exclusion rectangle provided by the calling application
- The pen leaving the proximity of the tablet
- A timeout on new pen data

After recognition is completed, the recognizer calls back into the RC Manager to process the results. It is at this point that dictionary processing ensues. Note that the **Recognizer** function has called back into the RC Manager before returning from **RecognizeInternal** function.

Recognition occurs concurrently with writing. The pen driver reports the points to the RC Manager at interrupt time, and the recognizer operates independently in the interim. After the user finishes writing, there is very little delay before the results are displayed on the screen.

## Dictionary Processing

The dictionary path provides a means to check a recognition result against an expected or preferred set of results. Most recognizers, including the Microsoft recognizer, return alternatives along with their best-guess result. The method by which semantic or language

knowledge can be applied to modify results—that is, to decide if one of the alternatives is preferable to the best guess—is to pass a data structure that embodies the notion "best guess plus anything else remotely possible" to a dictionary capable of making this determination. The dictionary then decides if any of the alternatives should replace the best guess.

A Pen Extensions dictionary is a DLL with a predefined set of exported functions. The RC Manager can use **LoadModule** and **GetProcAddress** to call the required functions at run time. The concept of a dictionary and its capabilities are loosely defined in the Pen Extensions architecture. They might be relatively smart, with lots of contextual intelligence, or relatively dumb, looking up words and replacing a best guess with an alternative. The API is flexible enough to support either.

The RC Manager passes a single result to a chain of dictionary DLLs until one of the dictionaries decides to make a correction in the results. Once a single dictionary has determined that it knows the results should be corrected, no more dictionaries are called; the results are bundled up and sent to the application that called **Recognize** function with a WM_RCRESULT message.

## The Results

The recognition results are passed back to an application by a new message, WM_RCRESULT. The message contains a pointer to a data structure that contains the ink the user has entered, the recognizer's best guess, the bounding rectangle of the input, and the list of possible alternatives. The application can process this information however it decides to do so.

All of this occurs before the **Recognize** function terminates. In fact, the application is several functions deep at this point, as illustrated in Figure 2.4.

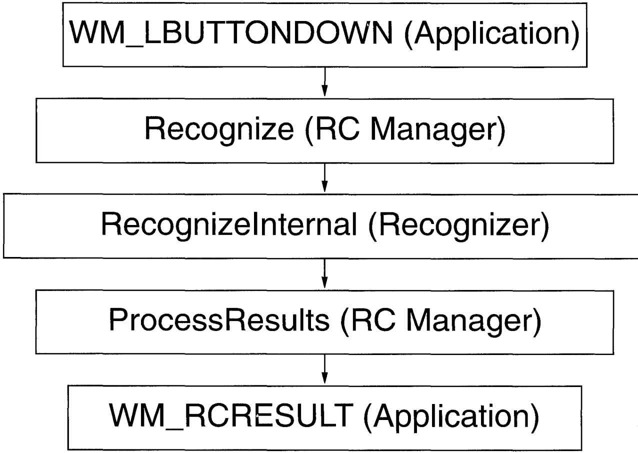



Figure 2.4. Data flow, LBUTTONDOWN to WM_RCRESULT

After an application returns from the WM_RCRESULT message, the call tree is unwound, with some cleanup occurring at each step. The **Recognize** function returns, and the remaining WM_LBUTTONDOWN processing, if any, is carried out by the application.

One final note: no WM_LBUTTONUP is received by an application that has called **Recognize** function. The RC Manager removes this message from the queue and does not allow it to be passed on to the application. Application designers need to take note of this.

## The Pen Message Interpreter and the Rest of the System

The cursor is an I-beam when over an area designated for textual input and management. When detected, the cursor changes into a pen cursor. When the pen goes down, the Pen message interpreter creates an invisible window that overlies the entire screen.

This invisible window serves as the agent for pen-ignorant applications. Inking actually occurs on this window. It is this window that interacts with the Pen API to perform recognition. The WM_RCRESULT message is sent to the invisible window, which then maps the results to keystrokes and mouse messages. The WM_CHAR, WM_MOUSEMOVE, and WM_LBUTTONDOWN messages that correspond to the gesture or text entered are entered into the system at the lowest level—by **Keyboard_Event** and **Mouse_Event**, respectively. The Pen message interpreter serves as both a logical keyboard and a logical mouse.

After the events are posted, the invisible window is destroyed. The results are that interaction with the pen is possible, all of the gestures function as expected, and text can be inserted at the insertion point. This use of an invisible window is present in the Windows Notepen application.

The Pen message interpreter makes it possible to use the pen with applications designed only for keyboard and mouse. However, if you design your applications with the pen in mind, you will increase their usability because they will support the entirety of the pen interface.

## The Gesture Macro Layer

An additional layer of functionality not illustrated in Figures 1 through 3 is the Gesture Macro Layer.

The Gesture Macro Layer is a system service that functions similarly to keyboard macro layers, in which the binding of a keystroke to an action implies that the keystroke will not be available to applications. If you bind a circle-letter gesture to an action, then that gesture will not be available to applications.

Figure 2.5 illustrates the functionality of the Gesture Macro Layer.

When the recognizer recognizes a circle-letter gesture, it passes the result to the Gesture Macro Layer to determine what should be done with it. There are three possible outcomes: no gesture binding, gesture binding that contains only printable characters, and gesture binding that contains nonprintable characters.

## No Gesture Binding

If there is no gesture binding, the WM_RCRESULT message is passed on to the application. The behavior is as if there were no Gesture Macro Layer.

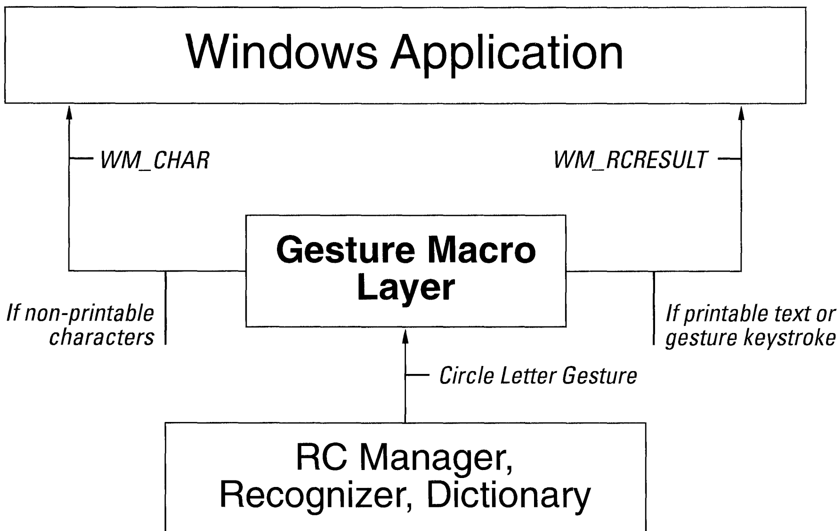



Figure 2.5. The Gesture Macro Layer

## Gesture Binding; Binding Contains Only Printable Characters

If a circle-letter gesture is bound only to printable characters—a common case—then the printable characters are returned to the application by WM_RCRESULT as the recognizer's best guess. The alternatives are still available, but the best guess is the user-provided character mapping for the circle-letter gesture. There are two important consequences of this action:

- The application gets a result that bears no resemblance to the ink the user entered or the set of alternatives suggested by the recognizer. Specifically, the ink will be that for a circled letter, and the result may be a very long string. A flag is set in the RCRESULT structure indicating that the Gesture Manager has made a replacement.
- The positional information associated with the gesture is retained. Because there has been no translation to WM_CHARs, and the entire set of results information is available to the application, any position dependence associated with the gesture is maintained. This ensures that, for those applications that attach importance to the position of input, the maximum level of this information is retained in results from the RC Manager.

## Gesture Binding; Binding Contains Nonprintable Characters

It is possible to bind a gesture to invisible or nonprintable characters such as ALT, CTRL, and F1. For example, the circle-s gesture might be bound to ALT+F+S (the Save command)

in the majority of Windows applications. If a gesture binding has characters of this nature, it cannot simply be specified as the recognizer's best-guess result, because the recognizer's best guess is always printable.

The Pen Extensions are designed so that the values returned from a recognizer map only to printable items. Recognizers return a special 32-bit value for each symbol recognized. The nonprintable characters are not represented in the space of acceptable 32-bit recognizer symbol values.

Because there is no way to return nonprintable characters with the WM_RCRESULT message, the characters must be sent to the application as WM_CHARs. There are two subcases for a nonprintable character binding: the characters represent keyboard shortcuts for editing gestures, or they are random.

### Nonprintable Characters As Keyboard Shortcuts

In this case, the nonprintable characters represent the keyboard shortcuts associated with standard editing gestures. Examples are SHIFT+DEL for the Cut command and SHIFT+INS for the Paste command. The Gesture Macro Layer replaces the circle-letter gesture result with the corresponding standard editing gesture result.

It is important to note that the standard keyboard shortcuts are logically reserved by the Gesture Macro Layer. Consider the scenario in which an application uses SHIFT+DEL for something other than the Cut command. If the user maps a circle-letter gesture to SHIFT+DEL, the Gesture Macro Layer will not map the circle-letter gesture to SHIFT+DEL—it will map it to the cut gesture. The application will never see SHIFT+DEL—it will see Cut.

### Nonprintable Characters As Random Characters

In this case, the nonprintable characters are just random and uninteresting nonprintable characters sent to the application as WM_CHAR messages. The keystrokes are entered into the Windows system through the **Keyboard_Event** function API in KERNEL. In other words, the Gesture Macro Layer becomes a logical keyboard driver.

# The Recognition Process

This chapter describes the recognition process in Microsoft Windows for Pen Computing. It discusses the primary data structures and the methods used by application programs to process recognition results. This chapter assumes that you have read the previous chapter, "The Architecture of the Pen Extensions."

Further details about the structures, functions, and constants used by recognizers are contained in Chapters 9 through 11.

## Overview

Figure 3.1 highlights the steps necessary to produce recognition. These steps begin after an application has initiated the recognition process with a call to the **Recognize** function, for example.

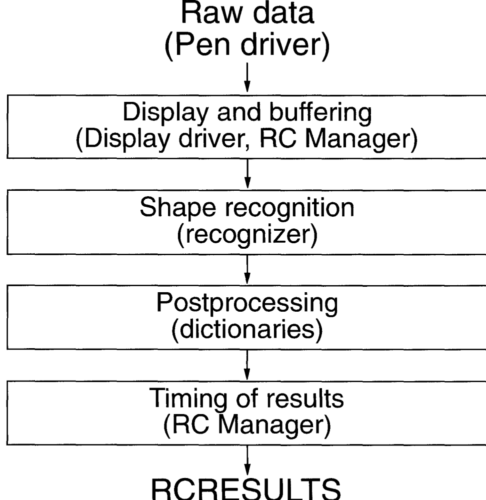



Figure 3.1. The recognition process

The basic process of recognition takes place as follows:

- **Display and buffering.** The RC Manager takes raw data from the pen driver and displays it as ink on the screen. The display of ink is incidental to the recognition process—it is required only to give the user feedback equivalent to that of a pen on paper. As the ink is displayed on the screen, it is being buffered for use in the next step of recognition.
- **Shape Recognition.** In this part of the process, the recognizer turns raw pen data into a predefined symbol such as a Roman character, geometric figure, or Kanji character. This phase begins with a recognizer callback into the RC Manager to query the buffered data. A recognizer processes the data in convenient chunks and determines the most likely set of symbols associated with the ink.
- **Postprocessing.** The postprocessing facility provides a measure of contextual information that can be used to improve recognition accuracy. For example, a typical postprocessing facility compares the word that was recognized against a set of expected results (some word list—for example, a dictionary).
- **Timing of Results.** During this phase, the results are returned to the caller in the method requested.

Shape recognition is often obstructed by the inability of the recognizer to resolve ambiguities in user input. Suppose, for example, the following text is passed as raw data to the recognizer.

Figure 3.2. Text sample

If you are familiar with the English language, you associate this ink with the word "kit." However, a recognizer might read it as the text string "lcit." In fact, the recognizer has little information to distinguish these two possibilities. At first glance, you might expect that the spacing of the letters could be used to make a determination. However, it's quite possible that the first two characters are indeed "l" and "c," accidentally written close together. For this particular ink, a recognizer might return the result that the input has a 40 percent chance of being "kit" and a 60 percent chance of being "lcit."

A postprocessing of these results would replace the possible "lcit" result with the word "kit," because "lcit" does not appear in an English-language word list.

An application can request that the results of recognition be returned on character, word, or line boundaries. An application can also request that the results be returned only when the entire recognition process is complete. The RC Manager passes results back to the application at the requested intervals, thereby completing the final step of the recognition process.

# RC: The Principal Data Structure

The **RC** structure is the primary data structure used in the recognition process. The following paragraphs discuss all of the various options available with the elements of this structure, listed "chronologically" with respect to the flow of information from ink to recognition.

## Using and Modifying Ink

The **nInkWidth** and **rgbInk** field of the **RC** structure specify the width and color of the ink left by the pen as the user writes with it. The **nInkWidth** field is an integer—specified in display coordinates (pixels)—that describes the width of the ink to draw. The **rgbInk** field is a DWORD that contains an RGB value used to render the ink on the screen. If the requested color does not map exactly to a possible color, the **GetNearestColor** function is used internally to determine the nearest solid color to use. The ink is drawn only in solid colors for the sake of speed.

## Ending Recognition: Pen State

When you design an application, you must consider when the recognition process will be considered complete. For example, the type of input expected will influence the criteria for ending recognition. In the case of a system gesture that consists of a single stroke, recognition ends after the user enters one stroke. In the case of a freehand drawing application, recognition might end when the pen leaves a bounding rectangle area where the user draws with the pen.

The standard way to end recognition is to wait for a pen time-out. The system waits until a specific interval has elapsed without new data from the pen, and then it stops recognition. The termination conditions are set in the **lPcm** bit field (the Pen Collection Mode field) in the **RC** structure. You can combine the PCM_ values discussed in the following paragraphs with the OR operator to create any termination condition you want to specify.

### PCM_PENUP

This value ends recognition as soon as the pen is lifted from the tablet surface. If you use this option, you limit user input to a single stroke before recognition results are returned to the caller. This is useful for a gestures-only field or for an application that performs some special action on single strokes of input.

One special use of PCM_PENUP might be for an application to implement scrolling in applications through flicks of the pen. Another might be for an application that recognizes only system gestures within a given screen region.

### PCM_RANGE

This value ends recognition as soon as the pen leaves the range of the tablet's proximity-detection mechanism. This option makes sense only for those tablet devices that support proximity detection. Use PCM_RANGE if you want recognition to end after a single letter of input or as soon as the user stops writing.

One difficulty with PCM_RANGE is that the control of proximity is set by the hardware and is not accessible to software control with the Pen API extensions. If you are going to use this termination condition, you should test it extensively on the target hardware to make sure that normal user variances in writing technique do not result in undesirable behavior such as premature attempts at recognition.

### PCM_RECTBOUND

This value ends recognition with the next pen down event that occurs outside a specified bounding rectangle. Use this option to keep the pen in inking mode for long periods of time. For example, you can use this option to keep the ink displayed as ink until the user lifts the pen and taps outside a field. Note that writing can extend beyond the bounding rectangle; it is the first tap outside this area that ends recognition.

You can also use this termination mechanism to limit the inking area to a window's client area. The rectangle to serve as the bounding rectangle is in the **rectBound** field in the **RC** structure. This field is a **RECT** structure that specifies the bounding rectangle in screen coordinates, unless the RCO_TABLETCOORDS flag is set. If this flag is set, the rectangle will be in tablet coordinates (thousandths of an inch).

### PCM_RECTEXCLUDE

This value ends recognition as soon as a pen down occurs within a specified rectangle. For example, you might use PCM_RECTEXCLUDE to create a button on the display that the user taps to end the recognition. The rectangle is specified with the **rectExclude** field in the **RC** structure. This rectangle is also in screen coordinates unless the RCO_TABLETCOORDS flag is set to specify tablet coordinates.

Note that PCM_RECTEXCLUDE has a higher priority than PCM_RECTBOUND if both are set and their specified rectangles intersect.

### PCM_TIMEOUT

This value ends recognition if there is no pen activity for the specified time. The **wTimeOut** field of the **RC** structure specifies how long (in milliseconds) to wait before the time-out occurs. A value of zero never times out.

PCM_TIMEOUT provides the most common method for ending recognition. It provides for a fairly natural process of printing, waiting for recognition, and then continuing with more text entry—much like a user's natural tendency to pause for thought when writing. This quality makes the time-out option a natural choice for recognition termination.

## The Application-Recognizer Connection

The accuracy of recognition can be improved if the application and recognizer communicate the type of input expected from the user. Such prerecognition contextual information can be used by a recognizer to improve recognition.

There are several categories of information that an application can provide to the recognizer to improve its accuracy. The following pages describe these categories of information, as well as the **RC** structure fields and values that can be used.

## Writing Location

It is likely that applications will provide some context to help users write neatly and achieve high levels of recognition. This might mean, for example, drawing lines on the screen that serve as guides for input—in effect representing ruled paper on the screen.

Another example is the bedit window class, which provides a boxed edit control based on a letter guide like the one illustrated in Figure 3.3. The letter guide improves recognition accuracy by constraining letters to individual cells.

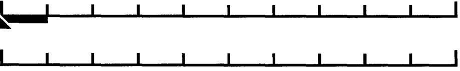


Figure 3.3. Letter guide of the bedit window class

The cells of the bedit class provide writing areas for the user. The **GUIDE** structure, an element of the **RC** structure, describes the grid of letter guides that the recognizer can use to separate the ink the user draws.

Suppose the "lcit" of Figure 3.2 is drawn in a bedit-class window. The ambiguous "lc" portion of the ink appears in one cell. With this additional clue, the recognizer can make the correct decision, recognizing the ink as a "k."

A similar use of the **GUIDE** structure is to resolve the ambiguities of some uppercase and lowercase letters such as "c/C," "s/S," "u/U," "o/O," "w/W," and "k/K." The recognizer is capable of independent processing to help make a determination, but it can be helped along by any information from the application regarding the baseline and midline rules provided to the writer.

An application can use the **wRcOrient** field to inform the recognizer of the digitizer's orientation.The coordinates returned from the digitizing device never actually change—they are just interpreted differently. As an example, this field can be used to enable writing along the vertical axis of a chart in a graphics program.

The **wRcDirect** field is used to specify the primary and secondary directions for input. It is not necessary for a recognizer to support this capability; the Microsoft recognizer, for example, does not. The **wRcDirect** field reflects the fact that most languages have a primary and a secondary writing direction. For English, the primary direction is from left to right, and the secondary direction is from the top down. For Chinese, the primary direction is from top to bottom, and the secondary direction is from right to left. Depending on the language, this information can be important for the recognizer.

## Specifying the Recognizer and the Type of Input Expected

Applications can supply additional information in the **RC** structure to specify which recognizer to use and what data to expect. Both types of information are supplied by fields in the **RC** structure.

The **hrec** field in the **RC** structure is a handle to the recognizer to use. Generally, an application will load recognizers with a call to **InitRC**, which generates a default **RC** structure. This loads the default system recognizer. Additional recognizers (such as a shape recognizer) are loaded with a call to **InitRecognizer.**

If the **hrec** field of the **RC** structure is NULL, then no recognizer is called. A NULL recognizer is used, for example, if the only intended result of the recognition event is to display and store the ink entered by the user.

In addition to specifying the recognizer, an application can use three additional clues to improve recognition accuracy: the characters expected by the recognizer, the priority to be assigned to different character sets, and the language to be used.

## Expected Characters

In many situations, especially those involving the fields in a form or dialog box, an application knows what type of characters the user is likely to enter. For example, a ZIP Code field should expect only numeric input—assuming it is configured exclusively for U.S. addresses. If you know the type of data expected, you can greatly improve the recognition accuracy by supplying this information to the recognizer.

You can specify the expected characters in the **alc** field of the **RC** structure as a 32-bit value. An application can combine the ALC codes with the OR operator to describe the expected input precisely. The alphabet, or ALC, codes are discussed in Chapter 11, "Pen Messages and Constants."

You can also set the **rgbfAlc** field to specify any subset of the ANSI character set as the set of expected characters. For more information, see the **RC** structure, discussed in Chapter 10, "Pen Structures."

## Priority

A second field that the application can use is the **alcPriority** field in the **RC** structure. This field gives a precedence rating to the possible ALC codes. The **alcPriority** field is set to a subset of the **alc** fields that specifies those results to be given the highest priority in making recognition decisions.

For example, an application could list ALC_UCALPHA | ALC_LCALPHA as the valid ALC codes indicating that the application is expecting only uppercase or lowercase letters. If you expect most users to enter uppercase versions of the letters, you set the **alcPriority** field to ALC_UCALPHA. This weights the recognizer toward the uppercase letters.

## Language

The **lpLanguage** field specifies one or more languages for the expected input. All characters within the ANSI character set can be specified by adding the RCIP_ALLANSICHAR flag to the **wIntlPreferences** field of the **RC** structure. It is often more useful to specify a subset of languages that should be enabled during recognition. The **lpLanguage** field points to an array of concatenated three-letter language codes that describe the current set of possible languages expected from the user.

By default, versions of MS Windows for Pen Computing will specify a single language relying on the International item from the Control Panel. Multiple language values might be set for European users or for other multinational pen platform users. By default, the values for this field are derived from the **sLanguage** element of the [Intl] section of the Windows WIN.INI file.

# The Timing of Recognition Results and Significant Events

The application can provide two different clues concerning the timing of recognition and the significance of events.

## Returning Results

The Microsoft recognizer can return recognition results at different intervals. The available intervals include word boundaries, stroke boundaries, character boundaries, new lines, and completion of recognition.

The timing of recognition is specified by the **wResultMode** field of the **RC** Structure. Results are returned no sooner than the requested interval, and they may be returned later. This may mean that even though an application requests results stroke by stroke, the results are actually returned when an entire word or an entire sentence is entered.

## Determining Significant Points

Pen events are stored in a buffer that is separate from the normal system queue of mouse events in Windows. This pen event buffer contains the high-resolution data from the digitizer device that is used by recognizers. Even though the two buffers are independent, it is possible to associate a particular mouse event with the appropriate pen event.

The pen computer submits each pen event to the Windows system as a mouse event. Along with the simple (x,y) screen coordinates, MS Windows for Pen Computing also passes a pointer that associates the mouse event with the pen event that generated it. This pointer is stored along with the queued mouse event in the Windows buffer. This buffer can be queried later by applications.

Use the **GetMessageExtraInfo** function to get the associated event pointer from Windows while processing a mouse message. For example, with a WM_LBUTTONDOWN message, an application should call **GetMessageExtraInfo** to get the pen event pointer. **InitRC** assigns a default value to **wEventRef**. Otherwise, it is the application's responsibility to assign the **wEventRef** field of the **RC** Structure before calling **Recognize**.

# Controlling the Recognition Process

There are several **RC** structure fields that affect how recognition proceeds. You can use these fields to implement special functionalities or manage special considerations during recognition.

The **clErrorLevel** field is a value from 1 to 100 that represents the percentage probability that a particular recognition result will be correct. The error level is the level below which a recognition result is considered to be unrecognized. The application can set the level at

which a recognizer will give up and return a "don't know" response rather than return an incorrect recognition result.

For example, you can modify this setting for a Social Security field where a very low error rate is required. By setting **clErrorLevel** to a high number, you can enforce a low error rate.

The **lpfnYield** field is a long pointer to a function supplied by the application that should be called by the **Recognize** function whenever it needs to yield the CPU for other background processing. If the **lpfnYield** field is NULL, the default Windows **Yield** function is called. It is the **lpfnYield** function's eventual responsibility to call the Windows **Yield** function. The default Windows function will not be called if **lpfnYield** is non-NULL.

The **wRcOptions** field is a bitwise combination of the RCO_ flags listed in the following paragraphs.

### RCO_BOXED

This value indicates that the **GUIDE** field of the **RC** structure contains valid data that should be used in the recognition process. RCO_BOXED suggests that boxes have been provided to the user for letter entry.

### RCO_DISABLEGESMAP

This value disables the Gesture Manager's ability to replace circle-letter gestures with one or more keystroke combinations. An application that has reserved the circle-letter gestures might consider using this flag to disable any user-provided meanings for those gestures, but this is not recommended.

The Gesture Manager is a macro layer, and, as with all macro layers, the user must understand that any input bound to a macro cannot be used in the context of another application, because that input will never reach the application. The macro will instead be mapped before it gets to the application. This behavior is identical for the Gesture Manager and any other keystroke macro recorder.

### RCO_NOFLASHCURSOR, RCO_NOFLASHUNKNOWN

These values disable the display of visual feedback to the user. Normally, if a recognition result is SYV_COPY or SYV_UNKNOWN, the user receives cursor feedback in the form of a brief change from the pen cursor to an I-beam cursor or a question-mark cursor.

### RCO_NOHIDECURSOR

This value prevents the cursor from being hidden while inking. You might use this option in a drawing package to provide adequate visual feedback with an opaque digitizer tablet. RCO_NOHIDECURSOR is rarely used with an integrated digitizer-display, because the user is pressing the pen down on the desired location directly.

### RCO_NOHOOK

This value prevents the system-wide recognition results hook from being called before the recognition results are passed on to an application. You would most likely set this option in an application using hooks that have re-entrancy problems. The application could then disable the hook recognition results in contexts that are likely to cause such problems.

### RCO_NOSPACEBREAK

This value informs the recognizer to send entire sentences, instead of individual words, to the dictionary. Doing this is useful if you're using a custom dictionary with special contextual or natural language parsing capabilities.

### RCO_SAVEALLDATA

This value specifies that all of the data points should be saved. In the default case, only those points used by the recognizer are saved. Other points might include pen up locations or pressure information.

### RCO_SAVEHPENDATA

This value is used to save the pen data. Normally, the ink returned to the application by MS Windows for Pen Computing is discarded after the application returns from WM_RCRESULT. If an application is to save the data, it must either copy it before returning from WM_RCRESULT or set this option.

Once RCO_SAVEHPENDATA is set, it is the application's responsibility to free the ink data.

### RCO_SUGGEST

This value tells the dictionary to make suggestions for a particular recognition result. In general, a dictionary can only promote a less likely alternative to be the first choice—it cannot make suggestions on its own as to the likely recognition result. RCO_SUGGEST enables the dictionary to make suggestions that are not necessarily among the alternatives supplied by the recognizer.

### RCO_TABLETCOORD

This value indicates that all coordinate values in the **RC** structure have already been converted to tablet coordinates. By default, all coordinate values in the **RC** structure are in screen coordinates.

## Specifying the User for Recognition

The final important values you can provide to the **RC** structure should be provided within the **lpUser**, the **wRcPreferences**, and the **hwnd** fields. In these fields, you can identify the user supplying the input and specify the window that should receive the recognition results.

The **lpUser** field points to the name of the current user. It is accessible from the Handwriting application in the Control Panel window. The current user name is the basis for all training and user-specific settings used by the recognizer—for example, left- or righthandedness and preferred time-out interval before recognition.

This information should be available at recognition time, because all training is inherently user-specific, and the left- or righthandedness information can be especially important for shape recognition. Because the Control Panel supports the addition of new users, multiple users can use the same machine with their individual preferences recorded in this field.

In conjunction with the **lpUser** field, the **wRcPreferences** field of the **RC** structure contains the current user's preferences. In the current version of MS Windows for Pen Computing, this contains information about the preferred writing hand and whether the recognizer should generate information for training.

The **hwnd** field of the **RC** structure indicates where results are to be sent. It is this window that receives the WM_RCRESULT message containing the results from recognition.

## Dictionary Processing

Several fields of the **RC** structure are used in the postprocessing phase of recognition. All of these fields affect the dictionary processing of recognition results.

The **rglpdf** field of the **RC** structure contains a list of entry points within various dictionaries that are used in the processing of recognition results. Recognition results are passed to each of these dictionaries in turn until one of them indicates that a correction has been made or there are no more entry points to be called.

In general, dictionary processing takes place word by word. Once a single correction is made, the results string is modified with the new result, and the dictionary processing code moves to the next word.

You can use the RCO_NOSPACEBREAK option to pass entire sentences on to dictionaries for correction. This option is not supported by the Microsoft-supplied dictionary.

The **wTryDictionary** field of the **RC** structure is a value from 1 to 4096 that specifies the number of alternatives in the symbol graph that will be passed to dictionaries. A standard value for **wTryDictionary** is 100. This value means that 100 alternatives (should that many alternatives exist) will be passed to a dictionary for possible correction.

The application can use **wTryDictionary** to trade-off performance for recognition accuracy since many dictionary calls are time consuming.

# Recognition Results: The RCRESULT Structure

This section focuses on the processing of recognition results, specifically the WM_RCRESULT message and the associated **RCRESULT** structure. The **RCRESULT** structure contains the actual recognition results that are sent from MS Windows for Pen Computing to a pen-aware application.

## The WM_RCRESULT Message

The WM_RCRESULT message is used to send **RCRESULT** structures back to applications. The **lParam** parameter points to the **RCRESULT** structure. An application must be prepared to receive this message before calling the **Recognize** function, because all WM_RCRESULT messages associated with a particular recognition event will be received before **Recognize** returns.

The WM_RCRESULT message can arrive more than once for a given recognition event depending on the frequency with which applications have requested that data be returned. Each message contains a pointer to a new, self-contained **RCRESULT** structure that contains the recognition results from the time of the last WM_RCRESULT message to the present.

The **wParam** parameter specifies the reason the message was sent—for example, end of recognition or because of some event like a word break. The **wParam** parameter contains one of the REC_ codes described in Chapter 11, "Pen Messages and Constants," describing why recognition has ended. These REC_ codes are the same as those returned by the **Recognize** function.

## Symbols and Symbol Values

MS Windows for Pen Computing defines a 32-bit space to hold recognition results. Out of this space, values are allocated for geometric shapes, gestures, letters of the alphabet, Kanji, Katakana, musical notes, electronic symbols, or any other symbols defined by the recognizer. For example, there is a unique 32-bit value associated with the letter "a."

These 32-bit values associated with various symbols are known as *symbol values* (SYVs). They are used internally by the Windows for Pen Computing functions to refer to recognized input. The application, however, uses the symbol graph for recognition results.

## The Symbol Graph

The first element of the **RCRESULT** structure, **syg**, contains the various alternatives for user input that are conveyed to the application. This data structure, called a symbol graph, contains all of the likely alternatives for the ink entered. Specifically, it is a directed graph of symbol values that describe the recognition event.

Consider the ink example presented earlier in this chapter, in which the word "kit" was confused with "lcit." The symbol graph representing this situation looks like the following:

```
{ lc | k } it
```

If the recognizer decides that the "k" is the likelier interpretation, the symbol graph looks like the following:

```
{ k | lc } it
```

Each **RCRESULT** structure contains a symbol graph that fully describes all of the results generated since the last **RCRESULT** structure was sent to the application. This means that in most cases, the symbol graph will contain all of the recognized input associated with an entire recognition event.

The symbol graph is not, strictly speaking, a graph of symbol values. It is possible to have more than a single symbol value occupying a place in the graph. For example, the possible meaning "lc" is two distinct symbol values occupying one place in the recognized input.

Additional information is needed to completely specify the letters associated with a set of ink. Further, each meaning is potentially a group of symbol values. The symbol graph must bind locations in the ink stream to multiple alternatives of one or more symbol values, each with associated probabilities, so that likelihood decisions can be made.

The symbol graph structure contains two additional data structures that provide this information: symbol correspondence structures and symbol elements.

The array of symbol correspondence structures (**SYC**s) delineates a specific chunk of ink out of the stream of ink entered by a user. Each **SYC** contains a first stroke and a last stroke. These two strokes and all of the strokes between them define the chunk of ink 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 map all of the ink associated with the **RCRESULT** structure. Note that the array of **SYC** structures will be generated only if the RCP_MAPCHAR flag is specified in the **wRcPreferences** field of the **RC** structure.

The second data structure used to delineate the ink and the characters is an array of symbol elements (**SYE**s). An **SYE** contains a symbol value, a confidence level (probability), and an index into the array of **SYC**s. There is a symbol element for every symbol value in the recognized input. Each has its own confidence level and pointer into the array of **SYC**s. Each **SYE** in the array is associated with its ink and the relative certainty with which it is recognized.

To determine that a single chunk of ink maps to more than one character, note that more than a single **SYE** structure maps to the same **SYC** structure. In other words, more than a single symbol value is associated with the same chunk of ink.

To determine which result is likelier, you can compare the confidence levels associated with the symbol elements.

The cost of any particular path through the symbol graph—"cost," that is, in terms of error level—is a measure of how poor a character match is and how much the recognizer must deviate from the pure character prototype to accept this match. The cost of a particular path is computed by adding the costs of all symbol values and dividing by the number of symbol values for a potential mapping. Thus, you can obtain the lowest-cost solution and provide it as the best guess.

Higher-cost solutions can remain in the symbol graph as alternatives. By postprocessing the results with dictionaries, you can then promote one of the alternatives to best-guess status if it is appropriate.

The symbol graph allows applications to generate choice lists accurately and precisely. It also allows applications to target input appropriately by illustrating the exact relationship between the symbols recognized and the ink entered by the user.

## The Best Guess

The **RCRESULT** structure also contains three fields that provide best-guess information. Together these fields describe the interpretation the recognizer and dictionaries will apply to the ink entered by the user, representing the best guess as to the user's meaning.

The **lpsyv** field is a string of symbol values that map to the best guess. The best guess can be in one of the three following states:

- It can be the lowest-cost path through the symbol graph, as discussed earlier. The **FirstSymbolFromGraph** function will generate such a **lpsyv** from a symbol graph.
- The **lpsyv** field can specify a path through the symbol graph that has been created by dictionary postprocessing—or perhaps a dictionary suggestion unrelated to the symbol graph. In this case, a dictionary has promoted one of the higher-cost paths through the symbol graph to the best-guess position. The only way to determine that this has occurred is to compare the lowest-cost solution, which is obtained with a call to **FirstSymbolFromGraph** with the **lpsyv** field.
- The **lpsyv** field may be the result of a Gesture Manager mapping. For example, the circle-letter gestures can be mapped to character strings. There are two ways to determine that this has occurred. You can compare **lpsyv** with the **FirstSymbolFromGraph** result, as discussed previously, or you can check the **wResultsType** field of the **RCRESULT** structure for the RCRT_GESTURETRANSLATED or RCRT_GESTURETOKEYS flag.

Additional information about the **lpsyv** field included in the **RCRESULT** structure is contained in **cSyv**, the number of symbol values in the **lpsyv** string, and **hSyv**, the handle to the memory block where **lpsyv** is allocated.

If the **lpsyv** field contains a string of symbol values associated with printable characters, you can translate the string of **SYV**s to a string of characters with the **SymbolToCharacter** function. This function will generate a normal C language string.

## Location and Position of the Input

There are several fields in the **RCRESULT** structure that provide information regarding the location and position of the ink entered by the user.

- The **nBaseLine** field is the recognizer's estimate of the baseline of the ink entered by the user. If the baseline is not known, this value will be zero. The Microsoft recognizer does not use this field, so it sets **nBaseLine** to zero.
- The **nMidLine** field is the recognizer's estimate of the midline of the ink entered by the user. If the midline is not known, this value will be zero. The Microsoft recognizer does not use this field, so it sets **nMidLine** to zero.
- The **rectBoundInk** field is a Windows **RECT** structure that contains the bounding rectangle that, in turn, contains the ink entered by the user. Typically, **rectBoundInk** is used to invalidate the area of the screen in which inking occurred, or otherwise to update the display in the appropriate location. This occurs, for example, when ink is replaced with recognized text.

When **rectBoundInk** is computed, the ink width is taken into account. It is also taken into account that this is the bounding rectangle for all of the data the user enters. It is not guaranteed that the **rectBoundInk** value in **RCRESULT** will be a subset of the **rectBoundInk** provided as the bounding rectangle for input.

## Contextual Information

Two elements of the **RCRESULT** structure provide information about the recognition event, but not as a part of the actual results of recognition. They are **lprc**, a far pointer to the **RC** structure passed in to **Recognize**, and **wResultsType**, a flag that describes how the recognition event actually proceeded. The values of the RCRT_ constants used in the **wResultsType** flag are described in Chapter 11, "Pen Messages and Constants."

## The Ink

The final two elements of the **RCRESULT** structure contain information about the ink entered by the user:

- The **pntEnd** element contains the last point of the ink data from the user only if PCM_RECTBOUND or PCM_RECTEXCLUDE have been specified.
- The **hpendata** element is a handle to a pen data memory block that contains all of the ink information entered by the user.

Applications that manage ink in any way will reference the **hpendata** field of **RCRESULT** extensively. Some of the functions an application can perform on ink include recognizing it, training it, scaling it, drawing it, adding points to it and getting points out of it, copying it, and saving it. These capabilities and the Pen API functions that provide them are discussed in Chapter 4, "Managing Ink in Pen Applications."

# Managing Ink in Pen Applications

This chapter introduces the MS Windows for Pen Computing ink data type and discusses its internal structure, the Pen APIs that manipulate ink, and some implementation scenarios involving ink in applications.

The use of ink falls into two broad categories:

- Ink can be left on the screen without any character recognition.
- Ink can be managed by an application when it is performing delayed recognition.

Ink that is left on the screen without recognition is the more important application. In this usage category, the ink and screen behave like an electronic notepad, but with a few important differences. The electronic ink can be copied, scaled, erased, stored, indexed, and otherwise manipulated in ways not possible with ink on paper. This ink capability is one of the strongest features of the new pen-based computing systems.

The second usage of ink management is by applications performing delayed recognition. Delayed recognition is the process of storing ink with no loss of accuracy so that recognition can be performed at some later time.

## The HPENDATA Data Type and Ink

Ink used in the Pen Extensions is accessed by **HPENDATA**—a handle to pen data. The pen data is a memory block. This data structure is analogous to the other Window "H" data types such as **HDC**, **HCURSOR**, and **HPEN**. With these the **HPENDATA** type shares certain similarities:

- The handles are references to data structures that reside somewhere in memory blocks.
- The handles to memory are passed to the Pen Extension API functions that perform useful work on the data structures.
- Applications should ignore the details of the underlying data structure and use the API functions alone to do the work.

However, some details of the **HPENDATA** type are of interest. The remaining part of this chapter discusses the data structure and the APIs used to manipulate it.

### The Basics

The **HPENDATA** structure is a block of memory made available by an internal call to the Windows **GlobalAlloc** (global allocation) function. Unlike most other Windows data types that are allocated out of the USER or GDI heaps, the size of the **PENDATA** blocks require that their allocation come from the Windows global heap.

The **HPENDATA** structure is flat—that is, it contains no references to memory locations, and all of the information is stored in a single, contiguous block of memory. This means that the data structure can be written to disk and read back in again without corruption due to invalid pointers. If an **HPENDATA** block is written to disk, all that is required to restore the **HPENDATA** is that the data structure be read back in again into an appropriately sized, globally allocated block of memory.

There is a 64K limit on the size of the memory block containing pen data. At standard report rates of 120 samples a 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, roughly two and one-half minutes of pen activity can be encompassed in a single **HPENDATA** structure.

# The Details

Figure 4.1 illustrates how the pen data memory block referenced by **HPENDATA** handles is laid out.

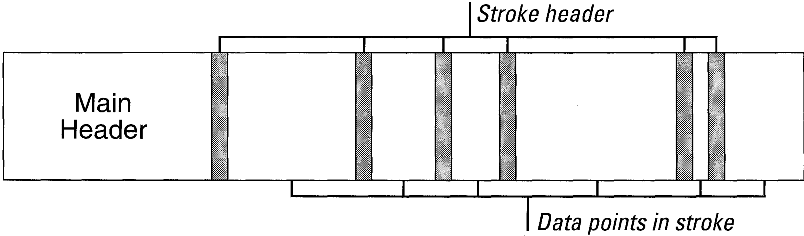



Figure 4.1. HPENDATA memory block

The layout of pen data in memory is a simple hierarchy. Data points are grouped by the strokes in the order in which they are entered. The **HPENDATA** block of memory begins with a descriptive header area.

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

## Data Points

The data points associated with each stroke are initially (x,y) coordinates in tablet coordinates with a resolution of one-thousandth of an inch and an origin at the upper-left corner of the tablet. Tablets are required to report points in this scale regardless of their actual resolution.

The Pen Extensions include functions to scale the points from the one-thousandth-inch resolution in the **HPENDATA** memory block to other metric systems. It is not necessary for the data in a **HPENDATA** memory block to remain in thousandths of an inch.

If Windows for Pen Computing is running in a portrait mode, the tablet is still required to report coordinates as if the current upper-left corner of the display is the upper-left corner of the tablet. The application does not need to concern itself with the current orientation of the screen. The (0,0) coordinate in Windows display coordinates is always equal to (0,0) in tablet coordinates.

Each (x,y) data point can be accompanied in the memory block by additional data such as pressure, angle, or rotation, assuming the tablet supports it. The main header section of the **HPENDATA** memory block describes how this additional information is stored in the stroke data areas for each data point.

Internally, any such OEM data is stored immediately following the block of (x,y) coordinates for a stroke. For each such coordinate there is a corresponding index in the OEM data block.

The ink in pen data structures has the absolute origin (0,0) set at the upper-left corner of the tablet, and not the relative (0,0) coordinate of the client window. This can cause difficulties when windows are moved around on the screen. If you encounter such difficulties, you'll need to modify your pen application. Your application should store the window-relative offset of the ink and constantly move the ink data to reflect the new location of the ink in client coordinates each time the ink is rendered.

Whenever a window is moved, the ink should move along with it and be repainted in a location relative to the upper-left corner of the window. One function you can use to accomplish this is the **OffsetPenData** function. This function immediately and permanently offsets all of the points in the pen data by an amount that you specify. If you offset them by the position of the client window, you can make the pen data relative to your client window instead of the tablet. In effect, this makes the ink position window-relative, but still in tablet coordinates.

## Stroke Header

A stroke is defined as the points that are recorded between the transition of the pen from down to up. The normal case is a stroke that occurs between the time the pen makes contact with the surface of the tablet and when the pen is lifted from the surface of the tablet. Some tablets and applications may also use proximity strokes for those points received when the pen is not in contact with the tablet.

Internally, the stroke header is a **STROKEINFO** structure (described in Chapter 10, "Pen Structures"). Each **STROKEINFO** structure corresponds to the pen's leaving or touching the tablet surface. Each of the pen events between these transitions is considered part of a single stroke.

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 maximum size of an **HPENDATA** memory block (64K).

The information in the stroke header includes the following items:

- The number of points in the stroke
- The number of bytes in the stroke (only used internally)
- The state of the pen during the stroke (up or down)
- A time index indicating when the stroke began

The time index is particularly useful if you need to reproduce the data on the screen exactly as the user entered it. For example, using the time index for the beginning of a stroke and the fact that the number of points per second is a known quantity, a signature can be retraced with the same speed used to enter it.

## Main Header

The first part of the main header of the **HPENDATA** data block is the **PENDATAHEADER** structure described in Chapter 10, "Pen Structures." The **PENDATAHEADER** contains the following types of information:

- The number of strokes
- The number of points in those strokes
- The remaining amount of memory in the memory block for new information
- The bounding rectangle of all points
- The ink color
- The ink width

This information is used by applications when they query portions of the pen data out of the **HPENDATA**, or render pen information to the screen.

The **wPndts** field of the **PENDATAHEADER** structure describes the state of the data in the **HPENDATA** block. The possible states include compression type, inclusion or exclusion of up points in the data, and an indicator of OEM data. The **wPndts** field can be used by applications to determine the state of the pen data, but it is most often used by the Pen APIs for this purpose. The **wPndts** element is a bitwise OR combination of the PDTS_* flags described in Chapter 11, "Pen Messages and Constants."

The final component in the main header is a **PENINFO** structure. Briefly, the **PENINFO** structure contains information on the tablet device where the data originated. This includes such information as the width, height, resolution, report rate, proximity capabilities, and barrel button status. For more information about the **PENINFO** structure, see Chapter 10, "Pen Structures."

The **PENINFO** structure also specifies if additional data is available beyond simple (x,y) coordinates.

If the **cbOemData** field of the **PENINFO** structure is greater than 0, there is more information available. The format and order of this extra information are contained in the **rgboempeninfo** array of the **PENINFO** structure. This array describes the order,

minimum value, and scale of any OEM specific pen data reported along with the (x,y) coordinate data.

For example, suppose this array has as its first index a pressure indicator and as its second index an angle indicator. This means that in the data point area of the pen data memory block, every (x,y) coordinate is associated with two bytes of pressure data and two bytes of angle data in the OEM data section of the pen data area. All applications should use the **PENINFO** structure to determine the nature of the data associated with each (x,y) location contained in pen data.

# The Ink Functions

The following sections describe the categories of functions that manipulate ink and the **HPENDATA** structure on behalf of applications. For complete details on these functions, see Chapter 9, "Pen API Reference."

## Rendering Pen Data

The most common use of ink by an application is to render the ink on either screen or page. The following two functions, **DrawPenData** and **RedisplayPenData**, are used for this purpose.

### DrawPenData

```
DrawPenData( HDC hdc, LPRECT lprect, HPENDATA hpendata)
```

This function renders the pen data to the specified device context using the Windows GDI polyline function. The current settings in the device context are used to render the data. For example, the current pen, pen color, and mapping mode are used when rendering the points in the pen data. Note that the color and width of ink stored in the pen data block are not automatically used to render ink.

In general, **DrawPenData** does not render to the screen data that exactly matches the data previously drawn by the display driver and PENWIN.DLL at interrupt time. This discrepancy results because the display driver and GDI polyline functions use different algorithms to draw the data. The possible difference is an "off by one" error that visually appears as a shifting of some pixels around the edges, depending on the rounding done by **PolyLine**.

You should use the **RedisplayPenData** function to render ink to the screen precisely as the user entered it.

### RedisplayPenData

```
RedisplayPenData( HDC hdc, HPENDATA hpendata, LPPOINT lpOrg,
                  LPPOINT lpExt, int nInkWidth, RGB rgbcolor)
```

**RedisplayPenData** uses the same algorithm as PENWIN.DLL to render ink onto the screen so that it looks exactly like the ink rendered originally by PENWIN.DLL and the display driver at interrupt time. There are two methods you can use to accomplish this:

- For each **HPENDATA** received, store the current origin of the window containing the ink in screen coordinates. Whenever that **HPENDATA** must be rendered, its origin will be placed in the *lpDelta* argument of **RedisplayPenData**. It is critical that the origin for every **HPENDATA** be stored. The ink received while the origin is the same can be merged into a single **HPENDATA**, but ink received after an origin change cannot be merged.
- Immediately upon receipt of the pen data, call **MetricScalePenData** to convert the pen data to display coordinates. Then use **OffsetPenData** to convert from display coordinates to client coordinates. This ensures that the ink data has the origin of the window in which it is to be drawn as its origin. If this procedure is carried out on all ink data received, numerous **HPENDATA** memory blocks can be merged together (up to the 64K limit for a HPENDATA memory block).

  The disadvantage of this second method is that there will be some degradation in recognition rates if the recognizer is particularly scale-dependent. If you have used this method, use the following function call:

  ```
  RedisplayPenData( hdc, hpendata, NULL, NULL, 1, 0L)
  ```

The algorithm used to render pen data at interrupt time uses a square pen brush rather than a round one for reasons of speed. For wide ink, this optimization of the GDI means that the end of the inked lines may appear blocky. If you don't want this, use **DrawPenData** instead of **RedisplayPenData** to render the ink.

# Transforming Pen Data

There are three functions that manipulate the scale of the pen data in the **HPENDATA** memory block: **MetricScalePenData**, **ResizePenData**, and **OffsetPenData**.

## MetricScalePenData

```
MetricScalePenData( HPENDATA hpendata, WORD wPdts )
```

This function converts pen data between any of the standard Pen Extensions mapping modes. The mapping modes specified by *wPdts* have the same scale as GDI mapping modes, but they do not have the same origin. In the Pen Extensions, the origin is the upper-left corner of the display or tablet.

This function will be used in applications that already rely on the use of a mapping mode and need to pass one of their standard device contexts to **DrawPenData**. Once the pen data points are scaled to the mapping mode of their application, the default device contexts can be passed to **DrawPenData**, and the ink will appear in the proper scale.

There are two important things to note about **MetricScalePenData**:

- Because of rounding errors, the scaling is not reversible when you move between mapping modes. Generally, this is important only if the ink is to be used for delayed recognition. The problem arises when you move from the standard ink scale (HIENGLISH) to a scale of lower resolution. Once this is done, you have lost some of the data from the higher-resolution scale. The recognition accuracy of **Recognize Data** may diminish, even if the data is converted back into HIENGLISH.

- The scaling is not perfect and will result in numerous shifting, “off-by-one” errors when the function renders scaled data.

### ResizePenData

```
ResizePenData( HPENDATA hpendata, LPRECT lpRect)
```

This function scales ink into arbitrarily sized rectangles.

**ResizePenData** exhibits the same weaknesses of the other scaling functions. Proportions of the rectangle are preserved, but rounding errors prevent the scaling process from being reversible. If the ink is scaled to a rectangle larger than the bounding rectangle in the pen data header, delayed recognition is generally not affected.

### OffsetPenData

```
OffsetPenData( HPENDATA hpendata, int dx, int dy)
```

This function offsets the (x,y) data for the points in the pen data memory block. The increments *dx* and *dy* are added to the (x,y) points. You can use this function to shift the location of the pen data without changing the scale of the data and thereby affecting recognition.

A common use for **OffsetPenData** is to shift the points in pen data to be in window-relative tablet coordinates. Pen data can be stored in this format and easily rendered at the proper location in a window by this method.

## Pen Data Housekeeping

There are three functions that perform housekeeping operations on pen data memory blocks: **DuplicatePenData**, **DestroyPenData**, and **CreatePenData**. They are standard operations similar to many Windows data types.

### DuplicatePenData

```
HPENDATA DuplicatePenData(HPENDATA hpendata, WORD gmemFlags)
```

This function is a copy routine for pen data. An application can quickly generate clones of its pen data blocks using this function. The `gmemFlags` parameter is a combination of desired GMEM_* flags used by the Windows **GlobalAlloc** function.

### DestroyPenData

```
DestroyPenData( HPENDATA hpendata)
```

This macro maps to the **GlobalFree( hpendata )** function. Because pen data memory blocks are simply chunks of memory allocated by **GlobalAlloc**, the **GlobalFree** function frees them.

### CreatePenData

```
HPENDATA CreatePenData( LPPENINFO lppeninfo, int cbOemData,
        WORD wPdtScale, WORD gmemFlags )
```

Use this function to create pen data memory blocks from scratch. If an application provides the **PENINFO** structure that will reside in the header, the real size of any OEM data to be stored along with each (x,y) coordinate, and the scale of the points in the pen data, it can create its own pen data memory block.

The `gmemFlags` should be either GMEM_MOVEABLE or GMEM_DDESHARE. This enables the memory block containing the pen data to move within the Windows global heap and to be passed to other applications.

## Pen Data Input and Output

The functions listed below retrieve (x,y) data from pen data memory blocks and add new data to the memory blocks.

### GetPenDataInfo

```
BOOL  GetPenDataInfo(HPENDATA hpendata, LPPENDATAHEADER lppendataheader,
LPPENINFO lppeninfo, DWORD dwReserved)
```

This function retrieves summary information from the pen data memory block. It is generally called before any of the other pen data input or output functions are called.

### BeginEnumStrokes, GetPenDataStroke, EndEnumStrokes

```
LPPENDATA BeginEnumStrokes( HPENDATA hpendata)

BOOL GetPenDataStroke( LPPENDATA lppendata, WORD wStroke,
    LPPOINT FAR * lplppoint, LPVOID FAR *lplpvOem, LPSTROKEINFO lpsi )

WORD EndEnumStrokes( HPENDATA hpendata )
```

These three functions operate on the pen data memory block on a stroke-by-stroke basis. An application must call **BeginEnumStrokes** before calling **GetPenDataStroke**, and it must call **EndEnumStrokes** when it has finished querying any stroke-level data.

**BeginEnumStrokes** uses **GlobalLock** internally to get a pointer to the location of the pen data block.

**GetPenDataStroke** returns pointers into the locked pen data memory block. An application requests that a certain stroke be returned, and a pointer is returned in the *lplppoint* argument that points into the *hpendata* memory block itself. The *lpstrokeinfo* buffer is provided by the application and is instantiated with information about the stroke pointed to by **lplppoint.*

Because pointers are returned that indicate the actual points in the pen data block, you could modify the pen data in place. However, this is not recommended. Strokes are packed

one after the other; therefore, it is questionable how much functionality an application would get from this, because the stroke size is unable to change.

Once an application has finished calling **GetPenDataStroke,** it must call **EndEnumStrokes**. This unlocks the memory block containing the pen data and invalidates any of the pointers returned by **GetPenDataStroke**. For this reason, you must take care never to use the pointers returned by **GetPenDataStroke** once **EndEnumStrokes** has been called.

## GetPointsFromPenData

```
GetPointsFromPenData( HPENDATA hpendata, WORD wStroke, int wPnt,
                      int cPnt, LPPOINT lppoint)
```

This function copies points from a pen data memory block in a manner more traditional than that used by **GetPenDataStroke** function. The application provides a buffer in which to hold a block of points, and then it indicates which stroke to retrieve the data from. An application can also request a certain block of points within a stroke. In that case, *wPnt* is the first point to retrieve, and *cPnts* is the number of points to retrieve.

Use **GetPointsFromPenData** to retrieve a specific point or block of points from a particular stroke. For example, use this function to digest the points in a pen data block a few at a time to avoid allocating a large block of memory to get all of the points.

**GetPointsFromPenData** can also be used to return the last point in a stroke if the *wPnt* argument is greater than the number of points in the stroke. In addition, if *wStroke* is larger than the number of strokes in a pen data memory block, the points returned will be from the last stroke.

A quick way to query the last point in a pen data memory block is to use large numbers for *wStroke* and *wPnt*. For example, any number that creates more than 64K of point data is high enough to guarantee that the point returned will be the last one in the pen data.

## AddPointsPenData

```
HPENDATA AddPointsPenData( HPENDATA hpendata, LPPOINT lppoint,
         LPVOID lpvOemData, LPSTROKEINFO lpsiNew)
```

This function appends new blocks of points onto an existing pen data memory block specified by *hpendata*. The *lpsiNew* parameter specifies a **STROKEINFO** structure that describes the new points, and *lpvOemData* describes any OEM data to be added.

The **STROKEINFO** structure might represent points that are in the same state as those that are currently in the last stroke of the pen data. Because such points do not indicate a state transition, they are appended to the last stroke. If the **STROKEINFO** structure indicates that the state transition between pen up and pen down has occurred, a new stroke will be created to contain the points passed by this function call. The number of points in the *lppoint* array is indicated by the *cPnts* element of the **STROKEINFO** structure.

The return value from this function is usually the same **HPENDATA** passed into it. Any other return indicates an error condition.

# Compressing Pen Data

Data compression of pen data is an important element of pen applications. Combining the high data rates of the pen digitizing devices with large amounts of inking created by the user results in large memory blocks of ink data. The Pen Extensions offer an application many different compression options, each with a set of advantages and disadvantages depending on the future use of the ink.

In general, there are three types of compression:

- Compression that removes redundant or otherwise useless data from the data structure. The resulting **HPENDATA** can be passed immediately to any delayed recognition function without any loss in accuracy.
- Compression that employs a more complex algorithm to compress pen data points. The resulting **HPENDATA** cannot be passed immediately to a delayed recognition function. However, data can be decompressed and then passed to a delayed recognition function with no loss of accuracy.
- Compression that employs an algorithm that compresses the data with some loss of information. The result is that the data can no longer be used for delayed recognition, but it is still suitable for display purposes.

### CompactPenData

```
CompactPenData( HPENDATA hpendata, WORD wTrimOptions)
```

The **CompactPenData** function is the primary function used to compress pen data. The *wTrimOptions* option is a bitwise OR combination of the PDTT_ compression options described in Chapter 9, "Pen API Reference."

# Display Resolution Compression

There is an additional compression method that can be used if the application is intended to render data only for display and not for later recognition. This method saves the largest amount of memory space, but at the cost of recognition accuracy.

First, use **MetricScalePenData** to convert the ink from tablet coordinates to display coordinates (that is, PDTS_STANDARDSCALE to PDTS_DISPLAY). Second, use the **CompactPenData** option of PDTT_COLINEAR to remove the duplicate and colinear points. These two steps will massively reduce the number of points in the pen data, because many high-resolution digitizer points are thrown away by the conversion. However, this loss means that the data may not be correctly recognized later.

The application can still choose to perform any of the other PDTT_* compression methods on the ink to compress pen data further, making the **MetricScalePenData** call to MM_TEXT the first step along a path to ink in its most highly compressed form.

# Common Scenarios Using an Ink Object

One common scenario is the display of a scribble of ink that the user enters in a region containing a number of other scribbles. This ink object will most likely be displayed in window class designed to support various inking functionalities. When ink is entered into this region, there are several steps that an application should undertake.

## Offset the Ink

The ink should be offset from the origin of the window. However, this object will render ink precisely as it was inked at interrupt time, and the ink will not be submitted for delayed recognition. To render the ink in a usable state, you should convert it to PDTS_DISPLAY coordinates and offset it to the origin of the window instead of the origin of the display.

Assuming the origin of the window containing the ink to be at screen coordinates (x,y), the following calls are required:

```
MetricScalePenData( hpendata, PDTS_DISPLAY );
OffsetPenData( hpendata, -x, -y ) ;
```

## Generate New HPENDATA Structures

New ink from the user will generate new **HPENDATA** structures. These should be merged into the ink storage mechanism already established for the window. There are two ways to do this:

- You can add the contents of the new pen data to an existing pen data. First, extract the points in the new pen data, using **GetPointsFromPenData**. Then add them to the existing pen data structure representing the contents of the window, using **AddPointsToPenData**.

  Remember that a pen data memory block data structure is limited to 64K, and before the points are extracted they must have the same scaling or offset performed on them as on any previous ink.

- Alternatively, you can create a higher-level application data structure—such as a linked list or an array—that stores a number of **HPENDATA** handles, rendering them all as needed. The inking sample provided in the Pen SDK shows how to do this.

## Render the Ink

The ink should be rendered as necessary. There are several reasons why the ink may not be rendered exactly as the user entered it:

- A mapping mode other than MM_TEXT is already in use, so the ink will have to be scaled anyway. This will produce differences.
- The use of a square brush in **RedisplayPenData** as opposed to the round one used by GDI and **DrawPenData** may cause distortion in the display, especially if the ink is wide.

- The ink was originally intended for delayed recognition, and the overhead necessary to do this and render the ink properly is insufficient.

If none of these factors is present, use **RedisplayPenData**.

## Allow for Resizable Ink Objects

It is likely that an ink object is going to be resizable, and this will probably involve scaling the ink within the rectangle. You can best implement this functionality by using the *lprect* argument of the **DrawPenData** function or the *lpExt* argument of the **RedisplayPenData** function to scale the ink into the window's DC at display time. Scaling this ink at display time preserves recognition accuracy.

You can physically alter the points in pen data to achieve the same effect, using **MetricScalePenData.** However, this will affect accuracy by altering the data points.

## Compressing Ink Objects

Another key issue to consider for the ink object is how the ink will be compressed before it is stored.

In general, you should use the maximum amount of compression possible, consistent with the intended use of the ink. If the ink is to be submitted for delayed recognition, you must use one of the compression methods that is completely reversible or otherwise not destructive of the ink data. If the ink is to be scaled and redrawn smoothly, then you need to ensure adequate resolution for the stored data to facilitate rendering at any size.

For example, if the ink data is compressed to display resolution, it will not scale bigger than its first size without loss of fidelity. If however, the ink is left at tablet resolution, it will scale much more smoothly because digitizers have a much higher resolution than display devices.

## Saving the Ink to a File

Saving the ink to a file is a trivial matter, because the **HPENDATA** structure is flat. The application should first use **GlobalSize** to determine the size of the **HPENDATA** block. Then the application should call **GlobalLock** to lock the pen data handle and then store the data. Later, a memory block sufficiently large to store the data can be allocated from the global heap with **GlobalAlloc**, and the data can be read back in to the block. When the new block of data is unlocked, a valid **HPENDATA** will result.

# A Sample Pen Application

This chapter explains how to write a simple pen application. The source code for this sample program is in the \PENSDK\SAMPLES\SREC directory. The PENAPP.* files make up the sample application. This directory also contains the files (SREC.*) for a sample recognizer. For a complete description of this recognizer dynamic-link library, see Chapter 7, "Replaceable Components: Recognizers and Dictionaries."

This chapter assumes that you are familiar with the architecture of the Pen Extensions and the basics of Windows programming. An overview of the architecture is discussed in Chapter 2, "The Architecture of the Pen Extensions."

## Overview of the PENAPP Application

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

In operation, the PENAPP application accepts pen input through the Input child window. Depending on the menu option, PENAPP sends the input to the system recognizer, the shape recognizer, or the sample custom recognizer (SREC).

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. The Info child window displays one of three things:

- 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 (up, down, left, or right). If there is no direction—that is, the pen is resting in contact with the tablet surface—the Info window displays a single dot.
- If the selected recognizer is the shape recognizer, the Info window displays a clean image of the shape. Recognized shapes are rectangles, ellipses, or lines.

The beginning of writing is signaled by a WM_LBUTTONDOWN message. The **IsPenEvent** function is called to confirm that this is a pen rather than a mouse event. Upon receiving this message, the program initializes the recognition context with a call to **InitRC**. In this application, the **rc.rglpdf[0]** field of the **RC** structure is set to NULL to disable dictionary processing.

After the recognizer has recognized pen input, the window procedure receives the WM_RCRESULT message. The *wParam* parameter of the message indicates the cause of the end of recognition (the REC_ code). The application tests to see if *wParam* is less than zero, indicating an error condition—for example, buffer overflow. If the pen input is

successfully recognized, the symbol value returned by the recognizer is copied to the PENAPP.C *syvGlobal* variable. In the default recognizer condition, the symbol value is also converted to an ANSI character.

The raw data is then copied with a call to the **CopyRawData** function. All of the window rectangles are invalidated and redrawn on the WM_PAINT message.

# WinMain and Initialization Functions

The PENAPP application uses **WinMain** and three initialization functions: **FInitApp**, **FInitInstance**, and **FLoadRec**. To an experienced Windows programmer, the **WinMain** and initialization functions will look very familiar.

## WinMain

The **WinMain** function is the entry point for a pen application, just as it is for any Windows application. In fact, the PENAPP **WinMain** function looks virtually identical to a regular Windows function, with just a few additions.

In a pen application, **WinMain** does the following:

- Declares an extern **HANDLE** for the recognizer, calls the initialization functions that register window classes, creates windows, and performs any other necessary initializations.
- Enters a message loop to process messages from the application queue.
- Stops the application when the message loop retrieves a WM_DESTROY message in **MainWndProc**. This termination includes unloading the recognizer with a call to the **UninstallRecognizer** function.

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
            LPSTR lpszCommandLine, int cmdShow)
  {
  MSG msg;

  extern HANDLE  hrecCur;

  lpszCommandLine;   // to prevent CS 5.1 compiler warning messages

  if (!hPrevInstance)
    {
    if (!FInitApp(hInstance))
      {
      return 1;
      }
    }

  if (FInitInstance(hInstance, hPrevInstance, cmdShow))
    {
```

```
    while (GetMessage((LPMSG)&msg,NULL,0,0) )
      {
      TranslateMessage((LPMSG)&msg);
      DispatchMessage((LPMSG)&msg);
      }
    }
  else
    msg.wParam = 0;

  return msg.wParam;
  }
```

# FInitApp

The **FInitApp** function is executed only once. It initializes the application data and registers the window classes. Following standard Windows practice, the function returns FALSE if it cannot register the window classes. Otherwise, **FInitApp** returns TRUE. The PENAPP child window classes are also registered in this function.

**FInitApp** has a pen-specific element. The cursor type, IDC_PEN, is the default cursor type required for a pen driver. The various pen cursors all begin with IDC_. For more information on them, see Chapter 9, "Pen Messages and Constants."

```
BOOL FAR PASCAL FInitApp(HANDLE hInstance)
  {
  WNDCLASS wc;
  HCURSOR hcursor;

  hcursor = LoadCursor(NULL, IDC_ARROW);

  /* Register Pen App window class */

  wc.hCursor = hcursor;
  wc.hIcon = LoadIcon(hInstance,MAKEINTRESOURCE(iconPenApp));
  wc.lpszMenuName = MAKEINTRESOURCE(menuPenApp);
  wc.lpszClassName = (LPSTR)szPenAppClass ;
  wc.hbrBackground = (HBRUSH)COLOR_APPWORKSPACE+1;
  wc.hInstance = hInstance;
  wc.style = CS_VREDRAW | CS_HREDRAW ;
  wc.lpfnWndProc = MainWndProc;
  wc.cbClsExtra = 0;
  wc.cbWndExtra = 0;

  if (!RegisterClass((LPWNDCLASS) &wc))
    return FALSE;
```

```
/* Register Pen App 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;
wc.hCursor = hcursor;
wc.lpszClassName = (LPSTR)szPenAppInfoClass;
wc.lpfnWndProc = InfoWndProc;
if (!RegisterClass((LPWNDCLASS) &wc))
  return FALSE;
wc.lpszClassName = (LPSTR)szPenAppRawClass;
wc.lpfnWndProc = RawWndProc;
if (!RegisterClass((LPWNDCLASS) &wc))
  return FALSE;
}
```

# FInitInstance

The **FInitInstance** function initializes all data structures for the program instance and creates the necessary windows.

**FInitInstance** looks like a standard Windows initialization function except that, after the windows are created, the function calls another initialization function, **FLoadRec**. This initialization function is required to install the sample recognizer used for handwriting recognition.

The default system recognizer is already installed. In fact, the procedure of loading recognizers is necessary only for custom recognizers; it is never required for the system recognizer.

```
BOOL FAR PASCAL FInitInstance(HANDLE hInstance, HANDLE hPrevInstance,
                       int cmdShow)
  {
  int   cxScreen = GetSystemMetrics(SM_CXSCREEN);
  int   cyScreen = GetSystemMetrics(SM_CYSCREEN);
  RECTrect;

  extern HWND   hwndMain;
  extern HWND   hwndInput;
  extern HWND   hwndRaw;
  extern HWND   hwndInfo;
```

```
hPrevInstance;    // to prevent CS 5.1 compiler warning message

/* Create Main window */

hwndMain = CreateWindow((LPSTR)szPenAppClass,
    (LPSTR)szPenAppWnd,
    WS_CLIPCHILDREN | WS_OVERLAPPEDWINDOW,
    0, 0, cxScreen, cyScreen,
    (HWND)NULL,
    (HWND)NULL,
    (HANDLE)hInstance,
    (LPSTR)NULL
    );

if (!hwndMain)
  {
  return FALSE;
  }


/* Create Input window */

GetClientRect(hwndMain, &rect);

hwndInput = CreateWindow((LPSTR)szPenAppInputClass,
    (LPSTR)szInputWnd,
    WS_CHILD | WS_BORDER | WS_CAPTION | WS_VISIBLE | WS_CLIPSIBLINGS,
    XInputWnd(rect.right), YInputWnd(0), DxInputWnd(rect.right),
      DyInputWnd(rect.bottom),
    hwndMain,
    nchildInput,
    (HANDLE)hInstance,
    (LPSTR)NULL
    );

if (!hwndInput)
  {
  return FALSE;
  }


/* Create Raw Data window */

hwndRaw = CreateWindow((LPSTR)szPenAppRawClass,
    (LPSTR)szRawWnd,
    WS_CHILD | WS_BORDER | WS_CAPTION | WS_VISIBLE | WS_CLIPSIBLINGS,
    XRawWnd(0), YRawWnd(rect.bottom),
```

```
        DxRawWnd(rect.right), DyRawWnd(rect.bottom),
        hwndMain,
        nchildInfo,
        (HANDLE)hInstance,
        (LPSTR)NULL
        );

    if (!hwndRaw)
      {
      return FALSE;
      }


    /* Create Info window */

    hwndInfo = CreateWindow((LPSTR)szPenAppInfoClass,
        (LPSTR)szInfoWnd,
        WS_CHILD | WS_BORDER | WS_CAPTION | WS_VISIBLE | WS_CLIPSIBLINGS,
        XInfoWnd(0), YInfoWnd(0),
        DxInfoWnd(rect.right), DyInfoWnd(rect.bottom),
        hwndMain,
        nchildRaw,
        (HANDLE)hInstance,
        (LPSTR)NULL
        );

    if (!hwndInfo)
      {
      return FALSE;
      }

    ShowWindow(hwndMain, cmdShow);
    UpdateWindow(hwndMain);

    return FLoadRec();   /*Load the recognizer  */
    }
```

## FLoadRec

The **FLoadRec** function uses the **InstallRecognizer** function to install a recognizer. The recognizer to be loaded is determined by the value of the application global flag, *miRecMode.* If *miRecMode* == *miSample,* this function installs the sample custom recognizer. If an application performs an explicit load on a default recognizer (as the following sample does), the application must also explicitly unload it. If an application is only going to use the system recognizer, there is no need to install it.

```
BOOL NEAR PASCAL FLoadRec(VOID)
  {
```

```
LPSTR lpRecogName;
HCURSOR hsave;

extern HWND    hwndMain;
extern HREC    hrecCur;
extern int   miRecMode;

/* hrecCur == NULL only at start of program */

if (hrecCur != NULL)
  {
  UninstallRecognizer(hrecCur);/* Unload any current recognizer */
  }

/* Install appropriate recognition DLL */

switch(miRecMode)
  {
  case miSample:     /* load sample recognizer */
    lpRecogName = (LPSTR)szSampleRec;
    break;
  case miShape:      /* load shape recognizer */
    lpRecogName = (LPSTR)szShapeRec;
    break;
  default:         /* load system recognizer */
    lpRecogName = NULL;
    hrecCur = NULL;
    return TRUE;   /* Don't need to load the Default recognizer */
    break;
  }
hsave = SetCursor(LoadCursor(NULL, IDC_WAIT));
hrecCur = InstallRecognizer(lpRecogName);
SetCursor(hsave);

if (!hrecCur)
  {
  MessageBox(hwndMain, "Could not install recognizer", szPenAppWnd, MB_OK);
  return FALSE;
  }

return TRUE;
}
```

# Data Handling and Display Functions

The **CopyRawData**, **DrawArrow**, and **DrawRawData** functions copy the raw data points, draw the arrow based on the symbol value returned by the custom recognizer, and draw the raw data in the Info child window, respectively.

The **CopyRawData** function saves a buffer of raw data points using an **HPENDATA** structure. A message box warns that **DuplicatePenData** is unable to allocate enough memory.

The **DrawArrow** function uses the value of the *syvGlobal* variable to determine which arrow to draw. The *syvGlobal* value is compared with the recognizer-specific symbols—for example, *syvEast*—defined in SREC.H.

The **DrawRawData** function draws the picture form of the input as taken by the recognizer. It uses the **GetPenDataStrokes** function to reconstruct the input.

The **DrawShape** function uses the value of the *syvGlobal* variable to determine which shape to draw. The *syvGlobal* value is compared with the recognizer-specific symbols (for example, SYV_SHAPEELLIPSE) defined in PENWIN.H.

The **SetGraphWindow** function responds to a request to change the recognizers. It sets the graph window, installs or reinstalls recognizers, and changes the menu setting, according to the *mi* value. The **SetGraphWindow** function uses the **FLoadRec** function to install a recognizer.

# MainWndProc

The **MainWndProc** function is the Windows procedure for the PENAPP application parent window. This is a standard Windows procedure.

```
LONG FAR PASCAL MainWndProc(HWND hwnd, unsigned message,
                  WORD wParam, LONG lParam)
  {
  LONGlRet= 0L;

  extern HPENDATA  hpendata;
  extern BOOL   fSaveData;
  RC rc;
  BOOL (FAR PASCAL *lpfnConfig) (WORD, WORD, LONG);

  switch (message)
    {
    case WM_COMMAND:
      switch (wParam)
        {
        case miExit:
          DestroyWindow(hwndMain);
          break;
```

```
    case miSample:
    case miShape:
    case miSystem:
      SetGraphWindow(wParam);
      break;

    case miSaveData:
      {
      HMENU hmenu = GetMenu(hwnd);

      CheckMenuItem(hmenu, miSaveData,
        (fSaveData = !fSaveData) ? MF_CHECKED : MF_UNCHECKED);
      break;
      }

    default:
      break;
    }
  break;

case WM_SIZE:
  {
  int x;
  int y;
  int DX;
  int dy;

  x = XInputWnd(LOWORD(lParam));
  y = YInputWnd(0);
  dx = DxInputWnd(LOWORD(lParam));
  dy = DyInputWnd(HIWORD(lParam));
  MoveWindow(hwndInput, x, y, dx, dy, TRUE);

  x = XRawWnd(0);
  y = YRawWnd(HIWORD(lParam));
  dx = DxRawWnd(LOWORD(lParam));
  dy = DyRawWnd(HIWORD(lParam));
  MoveWindow(hwndRaw, x, y, dx, dy, TRUE);

  x = XInfoWnd(0);
  y = YInfoWnd(0);
  dx = DxInfoWnd(LOWORD(lParam));
  dy = DyInfoWnd(HIWORD(lParam));
  MoveWindow(hwndInfo, x, y, dx, dy, TRUE);
  break;
  }
```

```
        case WM_DESTROY:
          if (hpendata)
            DestroyPenData(hpendata);

          if (hrecCur)     /* Unload current recognizer */
            UninstallRecognizer(hrecCur);

          PostQuitMessage(0);
          break;

        /*
          There is no reason to pass the WM_GLOBALRCCHANGE on to the Sample
          Recognizer, because it is a private recognizer and this application
          knows that the sample recognizer will not process the message.
        */
        case WM_GLOBALRCCHANGE:
          GetGlobalRC ((LPRC)&rc, (LPSTR)NULL, (LPSTR)NULL, NULL);
          if (hrecCur != NULL && (lpfnConfig = GetProcAddress(hrecCur,
"ConfigRecognizer")) != NULL)
            lRet = (*lpfnConfig) (WCR_RCCHANGE, 0, (LONG) (LPRC) &rc);
          break;

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

      return lRet;
      }
```

# InputWndProc

The work of a typical Pen application is illustrated in the **InputWndProc** function. This is the Windows procedure for the Input child window. You can use **InputWndProc** as a template for a typical pen-enhanced Windows procedure.

```
LONG FAR PASCAL InputWndProc(HWND hwnd, unsigned message,
                    WORD wParam, LONG lParam)
  {
  LONGlRet= 0L;
  RC    rc;

  extern HANDLE  hrecCur;
  extern int   miRecMode;
  extern HWND    hwndInput;
  extern HWND    hwndRaw;
  extern HWND    hwndInfo;
```

```
   extern char  szResult[cchMax];
   extern SYV   syvGlobal;


   switch (message)
     {
     case WM_LBUTTONDOWN:

       /* Two possibilities: user is using the mouse or the pen.
         If it is the pen, the user is starting to write.
         Initialize recognition context for recognizer */

       if (IsPenEvent(message, GetMessageExtraInfo()))
         {
         InitRC(hwndInput, &rc);

         rc.rglpdf[0] = NULL;   /* No dictionary search */
         rc.lRcOptions |= fSaveData ? RCO_SAVEALLDATA : 0;

         if (miRecMode != miSystem)
           {
           rc.hrec = hrecCur;
           }
         if(miRecMode == miSample)
           {
           rc.lPcm |= PCM_PENUP;  /* Set this flag for single strokes */
           }
         else
           rc.lPcm |= PCM_TIMEOUT;

         if (Recognize(&rc) == REC_BUSY)
           MessageBox(hwndMain, "Recognizer is busy", szPenAppWnd,
MB_OK|MB_ICONEXCLAMATION);
         }
       break;

     case WM_RCRESULT:
       {
       LPRECTlprect;

       /* The recognizer has recognized input and piped it through
         lParam (as an LPRCRESULT).

         The sample recognizer returns a symbol graph containing codes
         indicating the general direction the input stroke is written,
         according to the four compass directions.

         The shape recognizer returns a symbol graph indicating
```

```
          the geometric shape of the input (line, rectangle, or
          ellipse).

          The standard recognizer returns the recognized string.  */

        LPRCRESULT lprcresult = (LPRCRESULT)lParam;

        lprect = &(lprcresult->rectBoundInk);
        if ((int)wParam < 0)
          {
          syvGlobal = SYV_NULL;
          *szResult = NULL;
          lprect = NULL;   // set to null to invalidate entire
                    //  input window, because of possible overflow
          }
        else if ((lprcresult->cSyv != 0) && !(lprcresult->wResultsType &
RCRT_NOTHINGRECOG))
          {
          switch (miRecMode)
            {
            case miSample:
              syvGlobal = *(lprcresult->lpsyv);       /* Copy symbol value */
              break;
            case miShape:
              syvGlobal = *(lprcresult->lpsyv);       /* Copy symbol value */
              shapeRect = *(LPRECT)(lprcresult->syg.rgpntHotSpots);
              wLineStyle = (WORD)(lprcresult->syg.lRecogVal);
              break;
            default:
              *szResult = NULL;
              /* Set syvGlobal simply to pass test condition in InfoWndProc */
              syvGlobal = *(lprcresult->lpsyv);
              SymbolToCharacter(lprcresult->lpsyv, cchMax, szResult, NULL);
              break;
            }
          CopyRawData(lprcresult);
          }
        else// Nothing Recognized
          {
          syvGlobal = SYV_NULL;
          CopyRawData(lprcresult);
          }

        InvalidateRect(hwndInfo, NULL, TRUE);
        InvalidateRect(hwndInput, lprect, TRUE);
        InvalidateRect(hwndRaw, NULL, TRUE);
        break;
        }
```

```
    case WM_SYSCOMMAND:
      switch (wParam & 0xFFF0)
        {
        case SC_MOVE:    // Don't allow window to be moved
          break;

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

    case WM_PAINT:
      {
      HDC        hdc;
      PAINTSTRUCT ps;

      hdc = BeginPaint(hwnd, &ps);
      EndPaint(hwnd, &ps);
      break;
      }

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

  return lRet;
  }
```

# InfoWndProc and RawWndProc

**InfoWndProc** is the Windows procedure for the Info child window. The only pen-specific feature of this function is the code that determines the appropriate drawing technique to use, based on the recognizer in use. If the sample recognizer is used, **InfoWndProc** calls the **DrawArrow** function. If the default recognizer is used, a simple **TextOut** call is used.

Likewise, the **RawWndProc** is a normal Windows procedure for the Raw Data child window using no special pen-specific features.

# Using Pen Controls and the ProcessWriting Function

There are two techniques you can use for pen programming in addition to those described in Chapter 5, "A Sample Pen Application." The first technique replaces regular edit controls with pen-aware controls that recognize handwriting. The second technique uses the **ProcessWriting** function to process many of the details involved in recognizing handwritten input.

This chapter describes the structures of two simple pen programs that make use of these two techniques.

## Using the Hedit (Handwriting) Pen Control

The PENAPP program described in Chapter 5 uses a child window as the means of accepting handwritten input. An alternative means of input is to use a handwriting edit, or hedit (pronounced "h-edit"), control.

The hedit control is a complete replacement for the default Windows 3.1 edit control. The hedit control supports the handwritten input of characters and gestures, and it retains the standard edit control keyboard and mouse interface. An application can use an hedit control anywhere a regular edit control will work. An hedit control can also be used in dialog boxes.

A boxed edit, or bedit (pronounced "b-edit"), control is another type of handwriting control that provides letter guides for input. Bedit controls are discussed later in this chapter.

You create the hedit control with a call to **CreateWindow** using the hedit class. All of the regular edit control styles—for example, ES_LEFT and ES_PASSWORD—are supported.

The PENAPP application has to install the custom recognizer and handle the details of the recognition process. If you use the hedit control, these details are handled automatically.

The sample program in the file C:\PENSDK\SAMPLES\HFORM, included with Microsoft Windows for Pen Computing, shows how the hedit control can be used to build a pen-enhanced application. If you are unfamiliar with controls in general, consult your Windows programming documentation.

The HFORM sample program has several edit fields that are typical of a generic form application—name, address, city, and so on. The application registers itself as a pen-aware application so that the edit controls are replaced by hedit controls.

This application will run under MS Windows, version 3.0. When running under version 3.0, the edit control functionality is present. The boxed edit field and the picture field, however, are replaced by normal edit controls.

## Control Messages

Any control message (EM_*) that can be sent to an edit control can also be sent to an hedit control. In addition, a single new message, WM_HEDITCTL, has been added:

```
lRet = SendMessage(hwndEdit, WM_HEDITCTL, HE_xxx, lParam);
```

The *wParam* parameter indexes the message function, and *lParam* specifies values peculiar to the HE_* messages. For more information about the HE_ *wParam* and corresponding *lParam* values, see the entry for WM_HEDITCTL messages in Chapter 11, "Pen Messages and Constants."

In addition to having the basic characteristics of an edit control, an hedit control must make allowances for handwriting input. For example, the Delete 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 is reduced.

Likewise, restricting handwriting input to stay within the lines can also hinder recognition accuracy and user comfort. Rectangle offsets in the hedit control will compensate by making the writing area slightly larger than the client window size of a normal edit control. You use the HE_SETINFLATE and HE_GETINFLATE *wParam* values for this purpose, with the following **RECTOFS** structure:

```
typedef struct
  {
  int dLeft;
  int dTop;
  int dRight;
  int dBottom;
  }          // Rectangle offsets
RECTOFS FAR * LPRECTOFS;
```

An hedit window also processes tab and return gestures in a special way. If a user draws a tab or a return gesture in an hedit control that is in a dialog box, then the actual tab or return character is inserted into the text. The normal dialog behavior is preserved for TAB and RETURN keys—that is, TAB jumps to the next TABSTOP control, and RETURN jumps to DefPushbutton. The keystrokes can be made with either the actual keyboard or the on-screen keyboard.

An hedit window's parent receives the same notification messages (EN_*) as the parent of an edit window. The parent window receives a WM_COMMAND message. The *wParam* parameter contains the control ID. The *lParam* parameter contains the edit window handle in its low-order word and the message ID in the high-order word.

In addition, hedit controls also provide some HN_* notifications. These HN_ notification messages are described in Chapter 11, "Pen Messages and Constants."

## WinMain

The PENAPP.C sample program described in Chapter 5 focuses on some techniques for adding pen capabilities to a standard Windows program. For example, the **WinMain**, **FInitInstance**, and **HformWndProc** functions contain little pen-specific code. This

section will focus on the unique hedit features of HFORM.C. This chapter contains only the relevant sections of the code. For the complete program, refer to the sample code for HFORM.C.

**WinMain** uses the **RegisterPenApp** function to register the application to use hedit controls instead of edit controls. The **RegisterPenApp** function makes it possible to replace all edit controls in an application with hedits. This simplifies the task of making an application pen-aware and making that same application run under both Windows for Pen Computing and normal Windows.

To register itself as a pen-aware application, a program should use **RegisterPenApp**. The **RegisterPenApp** function should be called before any edit controls have been created. An application should unregister itself just before termination, using this same function.

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
            LPSTR lpszCommandLine, int cmdShow)
  {
  MSG msg;
  VOID (FAR PASCAL *RegisterPenApp)(WORD, BOOL) = NULL;

  lpszCommandLine;   // to prevent CS5.1 warning message
  if (!hPrevInstance)
    {
    if (!FInitApp(hInstance))
      {
      return 1;
      }
    }

  // If running on a Pen Windows system, register this application so all
  //  EDIT controls in dialogs are replaced by HEDIT controls.
  //  (Notice the CONTROL statement in the RC file is "EDIT",
  //  RegisterPenApp will automatically change that control to
  //  an HEDIT.)

  if ((hPenWin = GetSystemMetrics(SM_PENWINDOWS)) != NULL)
    {
    // We do this fancy GetProcAddress simply because we don't
    //  know if we're running Pen Windows.
    if ( (RegisterPenApp = GetProcAddress(hPenWin, "RegisterPenApp"))!= NULL)
      (*RegisterPenApp)(RPA_DEFAULT, TRUE);
    SetAppRecogHook = GetProcAddress(hPenWin, "SetRecogHook");
    }

  if (FInitInstance(hInstance, hPrevInstance, cmdShow))
    {
    while (GetMessage((LPMSG)&msg,NULL,0,0) )
      {
```

```
      // Check for menu accelerator message
      if (!TranslateAccelerator(hwndMain, hAccel, &msg))
        {
        TranslateMessage((LPMSG)&msg);
        DispatchMessage((LPMSG)&msg);
        }
      }
    }
  else
    msg.wParam = 0;

  // Unregister this app
  if (RegisterPenApp != NULL)
      (*RegisterPenApp)(RPA_DEFAULT, FALSE);

  return msg.wParam;
  }
```

# Initialization Functions

The **FInitApp** function initializes the application data and registers the window class. It is called only once, at the beginning of the program. It returns TRUE if successful.

The **FInitInstance** function initializes all data structures for the program instance and creates the necessary windows. A hook is set so that the application can trap recognition results before the hedit control receives them. The following code sets the hook:

```
  if (hPenWin && SetAppRecogHook)
    {
    // Set a hook so our app window can get recognition results
    //  before the HEDIT control does.  We do this for an
    //  accelerator gesture.
    if(!(fHookIsSet = (*SetAppRecogHook)(HWR_APPWIDE, HKP_SETHOOK,
            hwndMain)))
      {
      MessageBox(NULL, szHookErr, szHformWnd, MB_ICONSTOP|MB_OK);
      DestroyWindow(hwndMain);
      hwndMain = NULL;
      }
    }
```

The **FInitInstance** function also checks to see if the application is running under Windows for Pen Computing. If not, the delayed recognition is disabled. The code for this is the following:

```
  if (hwndMain != NULL && !hPenWin)
    EnableMenuItem(GetMenu(hwndMain), miDelayRecog, MF_GRAYED|MF_BYCOMMAND);
```

# HformWndProc

The **HformWndProc** function is the Windows procedure for the HFORM parent window. There are two particular things to note concerning the Pen extensions.

The **miDelayRecog** case highlights the use of the HE_ constants and the technique used to place the control in a delayed recognition mode. The program code for this case is the following:

```
case miDelayRecog:
  fDelayRecog = !fDelayRecog;
  ModifyMenu (GetMenu(hwndMain), miDelayRecog, MF_BYCOMMAND|MF_STRING,
          miDelayRecog, (LPSTR)(fDelayRecog ? szRecog : szDelay));
  {
  SetFocus(rgfield[0].hwnd);
  for (i = 0; i < cFieldsMax; i++)
    {
    if (rgfield[i].hwnd != NULL && rgfield[i].wFieldType==FIELDPIC)
      {
      if (fDelayRecog)
        // Place control in delayed recognition mode
        SendMessage(rgfield[i].hwnd, WM_HEDITCTL, HE_SETINKMODE ,
                (LONG)0L);
      else
        // Send message to HEDIT to recognize data
        SendMessage(rgfield[i].hwnd, WM_HEDITCTL, HE_STOPINKMODE ,
            (LONG)HEP_RECOG);
      }
    }
  }
  break;
```

**HformWndProc** also highlights the use of the WM_HOOKRCRESULT message to get recognition results and implement an accelerator gesture. The code for this is the following:

```
case WM_HOOKRCRESULT:    // getting recognition results
  {
  LPRCRESULT lpr = (LPRCRESULT)lParam;
  SYV syv;

  if((lpr->wResultsType&RCRT_GESTURE) &&
      (lpr->wResultsType&RCRT_ALREADYPROCESSED)==0)
    {
    syv = *(lpr->lpsyv);

    // This is an example of an accelerator gesture.  The
    //  user writes a circle with a 'D' (or 'd') inside,
```

```
        //  and we look for this gesture.

        // However, if the user through the gesture mapper has mapped
        // circle d to something else, then we don't need to do anything here.

        if(syv == SyvAppGestureFromLoAnsi('d')
          || syv == SyvAppGestureFromUpAnsi('D'))
          {
          PostMessage(hwnd, WM_COMMAND, miSampleDlg, 0L);

          // Let target window know result has already been acted upon
          lpr->wResultsType |= RCRT_ALREADYPROCESSED;

          lRet = 1L;
          }
        }
    }
break;
```

## FCreateForm

The **FCreateForm** function creates the hedit controls. The function returns TRUE if successful, FALSE if the hedit window cannot be created.

```
BOOL NEAR PASCAL FCreateForm(HWND hwndParent)
  {
  int   i;
  RC  rcin;
  LONG  lT    = GetDialogBaseUnits();
  int   cx    = LOWORD(lT)/2;
  int   cy    = HIWORD(lT)/2;
  RECTOFS rectofs;

  for (i = 0; i < cFieldsMax; i++)
    {
    PFIELD pfield = &rgfield[i];
    DWORD dwStyle = WS_VISIBLE | WS_CHILD | (hPenWin ? 0L : WS_BORDER) |
                    pfield->dwEditStyle;

    switch (pfield->wFieldType)
      {
    case FIELDPIC:
    case FIELDEDIT:
      /* Create Hedit window. */
      /* If running on a Pen system, this app will have
        been registered, so all EDIT controls will be changed to
        HEDIT controls */
```

```
  pfield->hwnd = CreateWindow(
      (LPSTR)"edit",
      (LPSTR)NULL,
      dwStyle,
      pfield->xEdit,
      pfield->yEdit,
      pfield->cxEdit,
      pfield->cyEdit,
      hwndParent,
      (HMENU)NULL,
      GetWindowWord(hwndParent, GWW_HINSTANCE),
      (LPSTR)NULL);

  break;

case FIELDBEDIT: // Comb field
  pfield->hwnd = CreateWindow(
      (LPSTR)(hPenWin ? "bedit" : "edit"),
      (LPSTR)NULL,
      dwStyle,
      pfield->xEdit,
      pfield->yEdit,
      pfield->cxEdit,
      pfield->cyEdit,
      hwndParent,
      (HMENU)NULL,
      GetWindowWord(hwndParent, GWW_HINSTANCE),
      (LPSTR)NULL);
  break;
}

if (!pfield->hwnd)
  {
  continue;
  }
```

The loop first loads the **PFIELD** structure with the appropriate window arguments. These values are defined in the static **FIELD** *rgfield* array, where one pen-specific feature is used. The ALC_ values signify the enabled alphabet to be used by the recognizer. The ALC_ values are described in more detail in Chapter 11, "Pen Messages and Constants." The values have the following meanings.

| ALC value | Meaning |
|---|---|
| ALC_DEFAULT | Recognize default alphabet |
| ALC_GESTURE | Recognize gestures |
| ALC_NUMERIC | Recognize numeric values |

Note that the ZIP field is specified as (ALC_NUMERIC | ALC_GESTURE), indicating that the recognizer should use the numeric values and gestures only. The recognizer will not need to distinguish carefully between lowercase "o," uppercase "O," and zero. All of these will be recognized as the numeric value of zero.

A **switch** statement is used to fill the various types of edit windows (hedit or boxed hedit). Boxed handwriting edit controls are explained in more detail later in this chapter.

Once the structure is filled, the hedit controls are created with a call to the Windows **CreateWindow** function.

The next step in creating the hedit controls is to set the Recognition Context (**RC**) preferences for each control. This step is optional.

```
/* Set RC preferences for this edit control */

if (hPenWin)
  {
  if (SendMessage(pfield->hwnd, WM_HEDITCTL, HE_GETRC, (LONG)((LPRC)&rcin)))
    {
    rcin.alc = pfield->alc;
    SendMessage(pfield->hwnd, WM_HEDITCTL, HE_SETRC, (LONG)((LPRC)&rcin));
    }
```

The Windows **SendMessage** function uses the new hedit control message WM_HEDITCTL with the HE_SETRC *wParam* value to set the **RC** preferences. The appropriate alphabet has been set by the preceding assignment statement. You can also set other **RC** preferences, such as ink color, at this time.

After the **RC** preferences are set, the default inflation rectangle offset is set so that it is one-half the base dialog unit for each respective axis. The HE_SETINFLATE *wParam* value specifies the adjustments to the client rectangle of the hedit window. If the control is not multiline, set the underline to on.

```
/* Change default inflation rectangle offset so it is
  half the base dialog unit for each respective axis. */

rectofs.dLeft = -cx;
rectofs.dTop = -cy;
rectofs.dRight = cx;
rectofs.dBottom = cy;
SendMessage(pfield->hwnd, WM_HEDITCTL, HE_SETINFLATE,
    (LONG)((LPRECTOFS)&rectofs));

// If no border, put under line in.
if ((pfield->dwEditStyle & ES_MULTILINE) == 0 &&
      pfield->wFieldType==FIELDEDIT)
  SendMessage(pfield->hwnd, WM_HEDITCTL, HE_SETUNDERLINE, (LONG)(1));
}
```

## Dialog Box Functions

The **SampleDialog**, **SampleDlgProc**, **ProcessFieldChange**, and **IFromHwnd** functions are used to process the dialog box information for the sample application.

The **SampleDialog** function opens a sample dialog box containing an edit (not an hedit) control. If the HFORM program is running under MS Windows for Pen Computing, then **RegisterPenApp** has been called previously. Because of this, the edit control will act like an hedit.

The **ProcessFieldChange** function contains the code to process cold recognition. Send an HE_GETINKHANDLE message to the control. If the LOWORD of the return code is NULL, the control is not in cold recognition mode, and the focus can be set. If it is in cold recognition mode, skip over it and check the next field.The *iCount* variable is used to break the loop if all fields are in cold mode.

```
iCount=0;
while (iCount<cFieldsMax && !wRet)
  {
  if (!LOWORD(SendMessage(rgfield[i].hwnd, WM_HEDITCTL, HE_GETINKHANDLE,
        (LONG)(LPSTR)&lInkData)))
    {
    hwndFocusField = rgfield[i].hwnd;
    SetFocus(hwndFocusField);
    wRet = TRUE;
    }
  else
    i = (i+inc) %(cFieldsMax);   // Calculate the next field
  iCount++;
  }
```

# Using the Bedit (Boxed Handwriting Edit) Control

The bedit (boxed edit) control is a special edit control that permits boxed display of text. The main advantages of a bedit control are improved recognition accuracy and the fact that the handwritten text goes where it is written.

A bedit control uses many of the same concepts that edit and hedit controls use—for example, selection, scrolling, and special keystroke assignments. Programming an interface for a boxed edit control is also similar to that for edit and hedit controls in the areas of control messages, notifications, and control styles.

Text in a bedit control is considered a single stream of text that is arranged in rows of cells for convenience. Text will always wrap around the rows without line breaks at word boundaries or returns.

You set the layout of a bedit control with the **BOXLAYOUT** and **GUIDE** structures contained within the **RC** structure. To set the **BOXLAYOUT** and **RC** structures, use the HE_SETBOXLAYOUT and HE_SETRC subfunctions of the WM_HEDITCTL message.

The structures are explained in detail in Chapter 10, "Pen Structures," and the message is described in Chapter 11, "Pen Messages and Constants."

By simply creating a window of class "bedit," you can create a boxed edit control with the default layout. For more information about the default dimensions of a boxed edit control, see the entry for BXD_ values in Chapter 11.

## Selecting Text

In bedit controls, there is no distinction between caret and selection. A single cell selection functions as a caret. Selection is displayed with a thick line under the selected cell. Boxed edit controls support selection gestures and keyboard selection methods.

The last selectable cell is one cell past the last non-empty cell in a boxed edit. The last selectable cell is also limited by the upper limit on the number of cells in the control, which is set by the EM_LIMITTEXT message or the size of a nonscrollable boxed edit.

## Word Wrap

Bedit controls do not support word wrap. The EM_SETWORDBREAK message has no effect on a boxed edit control.

## Scrolling

The bedit control supports only vertical scrolling. It responds to EM_LINESCROLL messages. It also supports WS_VSCROLL and ES_AUTOSCROLL window styles.

## Control Notifications

In addition to new HN_* notification messages, a bedit window's parent will receive the same notification messages (EN_*) as the parent of an edit window. The parent window receives a WM_COMMAND message. The *wParam* parameter contains the control ID. The *lParam* parameter contains the edit window handle in its low-order word and the message ID in the high-order word.

The HN_ notification messages are listed in Chapter 11, "Pen Messages and Constants." Both hedit and bedit controls use these messages.

## Control Messages and Styles

Any control message (EM_*) that can be sent to an edit control can also be sent to a bedit window with these exceptions: EM_FMTLINES, EM_SETREADONLY, EM_SETTABSTOPS, and EM_SETWORDBREAK. In addition, a single new message, WM_HEDITCTL, has been added:

```
lRet = SendMessage(hwndEdit, WM_HEDITCTL, HE_xxx, lParam);
```

The *wParam* parameter indexes the message function, and *lParam* specifies values peculiar to the HE_* messages. For more information about the HE_ *wParam* and the corresponding *lParam* values, see the entry for the WM_HEDITCTL messages in Chapter 11, "Pen Messages and Constants." Both hedit and bedit controls share many of these messages.

A bedit control supports all edit control styles with the exception of ES_AUTOHSCROLL, ES_CENTER, ES_LEFT, ES_RIGHT, and ES_READONLY.

## Using Bedit Controls in Dialog Boxes

A bedit control can be used in a dialog box in the same way as any other control, such as an edit or hedit control, except in the following cases.

If a FONT statement is present in the dialog template and the font listed in the statement is not available at the time the dialog is created, Windows picks another font and scales the dialog and the controls within it appropriately. Since the default dimensions of the bedit cell are based only on the system font, this scaling of the bedit control can cause the number of cells in the control to change slightly.

Applications that must have a specific number of cells in a bedit control can use one of the following techniques:

- Remove the FONT statement from the dialog template that would otherwise cause the dialog to use the system font.
- Select a font that is always likely to be available. The configuration of your Windows system determines which fonts are available.
- Resize the bedit control during the processing of the WM_INITDIALOG message to contain the required number of cells. You can resize a control with the **MoveWindow** or **SetWindowPos** Windows function. The default dimensions of bedit cells are specified by various BXD_ constants.
- Resize the dimensions of a bedit cell based on the size of the control. You can change the dimensions of a bedit cell using the **GUIDE** structure within the **RC** structure; use the HE_SETRC *wParam* value of the WM_HEDITCTL message.

A dialog box sets the font for all of the controls within it to the font selected for the dialog. This font size may turn out to be too small for use in the bedit control. An application should send a WM_HEDITCTL message with HE_DEFAULTFONT subfunction to the bedit control during WM_INITDIALOG message processing to get bedit to use a more suitable font. Alternatively, an application may also use the WM_SETFONT message on a bedit control.

# Using the ProcessWriting Function

The **ProcessWriting** function simplifies the task of converting an existing application to take advantage of handwriting input—both gestures and characters. **ProcessWriting** takes care of inking, removing the ink, and converting the results message to standard windows messages.

Depending on the existing code in an application, this function may or may not be suitable for making an application pen-enhanced. Although it simplifies pen programming, **ProcessWriting** does not provide some of the low-level flexibility available with other pen programming techniques.

After writing is completed, the ink is removed before any messages are sent to the window. After the screen is updated and the ink removed, the window receives a WM_RCRESULT message. If the application chooses to process this message, it should return a non-zero value.

If an application returns FALSE to the WM_RCRESULT message, the application receives the Windows messages shown in the following table. The messages are sent rather than posted. If the application returns TRUE to the WM_RCRESULT message, no further messages are sent.

| Name | Messages to hwnd |
|---|---|
| SYV_BACKSPACE | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture, followed by WM_CHAR of backspace. |
| SYV_CLEAR | WM_CLEAR. |
| SYV_COPY | WM_COPY. |
| SYV_CORRECT | WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_LBUTTONUP at the hotspot of the gesture, followed by WM_COPY; then the Edit Text dialog is activated, and it pulls text from the Clipboard. This uses the existing selection if any is present.<br><br>The previous contents of the Clipboard are lost. |
| SYV_CUT | WM_CUT. |
| SYV_CLEARWORD | WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_LBUTTONUP at the same point, followed by WM_CLEAR. |
| SYV_EXTENDSELECT | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture. The MK_SHIFT flag is set for the *wParam* of these messages. |
| SYV_PASTE | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture. WM_PASTE. |
| SYV_RETURN | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture, followed by WM_CHAR of RETURN. |
| SYV_SPACE | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture, followed by WM_CHAR of SPACE. |
| SYV_TAB | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture, followed by WM_CHAR of TAB. |

| Name | Messages to hwnd |
|---|---|
| SYV_UNDO | WM_UNDO. |
| Text | One WM_CHAR message per character of text. |
| All other results | No messages. |

The **ProcessWriting** function returns values less than zero if the application treats the event as a mouse event instead of a pen event. Values of less than zero are returned if the event did not come from a pen or the user did a press-and-hold action (REC_POINTEREVENT). A negative value is also returned in case of an error—for example, running out of memory.

## Modifying a Windows Program to Use ProcessWriting

A normal Windows program has the following basic structure:

```
ExampleWndProc(hwnd, message, wParam, lParam)
  {
  .
  .
  .
  switch (message)
    {
  case WM_ messages:
    code
  .
  .
  .
    break;

  case WM_LBUTTONDOWN:
    code for setting insertion point
    break;
  .
  .
  .
  case WM_CUT:
    code to remove text
    break;
    }
  }
```

You can modify this basic structure as follows to use the **ProcessWriting** function:

```
ExampleWndProc(hwnd, message, wParam, lParam)
  {
  .
  .
  .
  switch (message)
    {
  case WM_ messages:
    code
  .
  .
  .
    break;

  case WM_LBUTTONDOWN:
    if (ProcessWriting(hwnd, NULL) < 0)
  {
  Treat the event as if generated by a mouse. If REC_POINTEREVENT or
  REC_BUSY is returned, then begin a selection event.
  }
  break;
  .
  .
  case WM_CUT:
    code to remove text
    break;
    }
  }
```

The sample application for using the **ProcessWriting** function in a pen application is in the PENSDK\SAMPLES\PENPAD directory. The PENPAD.OLD source code is for the non-pen-enhanced version; PENPAD.C is for the pen-enhanced version that makes use of the **ProcessWriting** function.

# Replaceable Components: Recognizers and Dictionaries

This chapter describes the construction of recognizer and dictionary dynamic-link libraries (DLLs). Included with this SDK is the source code for a sample recognizer and dictionary that can form the basis for your own components.

# Recognizers

There are two broad types of recognition, characterized by type of input data: bitmap and vector.

Bitmap recognition uses optical character recognition (OCR) techniques, which accept an image of the characters to be recognized as input. This method of recognition is not supported by the Pen extensions application programming interface (API). However, it is possible to write an OCR application under Windows.

Vector recognition is the recognition of a sequence of pen-location data points collected as the pen is moving. This is the recognition model the Pen Extensions API supports.

To perform as the system default, a custom recognizer must, of course, conform to the Pen API. In addition, it should be able to do all of the following:

- Recognize at least one case of characters, punctuation, numbers, and predefined gestures
- Return images of prototypes for display purposes
- Associate raw data with matched results
- Return characters only within the requested subset
- Return an "I don't know" response
- Work with only (x,y) pairs as input data

## Converting Input to Usable Data

Three steps are involved in the process by which a recognizer converts input to usable data: processing the raw data of a user's entries, performing segmentation, and interpreting the order and direction of pen strokes.

### Processing Raw Data

Raw data for recognition consists of (x,y) pen coordinates. At a minimum, the pen data need be collected only while the pen is down. Optionally, additional pen data can be collected on a per-point basis. Such additional data may include, but is not limited to, pressure, height of pen above pad, angle of pen, and rotation of pen. Any recognition

module that relies on optional data will be limited to those platforms that provide pens and pen drivers capable of collecting this information.

The Microsoft recognizer requires only the (x,y) coordinates. The **OEMPENINFO** structure supports the collection of other pen data. For details, see the entry for **OEMPENINFO** in Chapter 10, "Pen Structures."

### Noise Reduction and Normalization

Noise reduction and normalization techniques can also be used. Noise reduction refers to filtering done to overcome limitations in hardware—for example, correcting pen skips or wild points. The pen driver is free to perform noise reduction.

Normalization refers to dehooking, deskewing, calculating baselines, correcting for baseline drift, removing redundant points, and interpolating. Normalization is internal to the recognizer, not the pen driver. During recognition, the recognizer must be able to return the raw data (as received from the pen driver) to the application, regardless of the type of normalization it performs internally.

## Performing Segmentation

Recognizers differ in their ability to separate individual characters within a stream. This is a crucial issue for recognizing different handwriting styles. The following table (derived from an IBM Research Report RC 11175, No. 50249, [5/21/85], "An Adaptive System For Handwriting Recognition," by C. C. Tappert) lists the forms of input, in decreasing order of constraint on the user.

| Input form | Definition |
|---|---|
| Boxed input | Each character must be written within its own box. |
| Discrete spaced | Each set of strokes can be identified as belonging to the same character by the surrounding space. (This is also called "external segmentation.") |
| Discrete run-on | Printed characters can overlap. |
| Cursive | Letters are connected by ligatures. Recognition is accomplished either by identifying discrete letters or by interpreting a whole word at a time. |
| Mixed | Discrete, run-on, and cursive writing are recognized. |

Figure 7.1 illustrates these various styles.

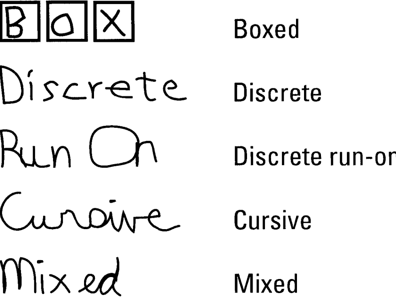



Figure 7.1. Handwriting styles

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

### Interpreting Stroke Order and Direction

The Pen API does not limit a recognizer's ability to deal with any of the following:

- Delayed strokes—for example, the cross on the "t" in "tree" if it was written after the "r."
- Correction strokes—for example, a little stroke put on the top of a "y" to make it look like a "g."
- Characters written out of order—for example, "cat" written as "c t" with the "a" filled in last.
- Variations in stroke order or direction—For example, a capital "E" with four strokes can be written in $2^4 * 4! = 384$ distinct ways.

## Returning Results

Recognition results should allow for multiple interpretations of input. For example, the word "clear" could be recognized as the following possible character sequence:

```
{ cl | d } ea { r | n }
```

This means the word is either "clear," "dear," "clean," or "dean." If multiple results are possible, the recognizer returns the possibilities listed in order of decreasing likelihood. If a recognizer has no internal concept of ordered guesses, it must impose an arbitrary order. In order for multiple recognizers to cooperate, a recognizer must have some concept of a really bad match and be able to return "unknown" instead of a best guess.

A recognizer should have some concept of confidence level in its matches, so that the user has some control over whether or not a matched character needs further processing. It is possible to run a recognizer that does not support "unknown" or confidence levels, but it will not work as efficiently with dictionary postprocessing, multiple recognizers, or additional processing.

For each symbol recognized, the recognizer must be able to associate the raw data that was applied to the match. It is not mandatory that all raw data points be associated with one and only one matched character; it is acceptable for the recognizer to ignore some input or assume that a given stroke is used to form multiple characters. The raw data will be used by applications to place the recognized text on the screen, redraw it, or send information to secondary recognizers.

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

## Results Message

A new message, WM_RCRESULT, has been added to Windows. This message is sent to the specified window when the recognizer has a result to return.

Any far pointers passed in this structure are valid only while processing the message. This is the application's chance to save the information about the raw data. After this message is sent, the recognizer can destroy its copy of the raw data.

The last WM_RCRESULT message for a recognition context is sent before **Recognize** returns and any other messages are sent to the application.

If the application returns 1, the recognition should continue. If the application returns –1, the recognition stops, and no more results are sent for this context. If the return value is 0, further processing of this message stops (used with the **ProcessWriting** function).

## Character Sets

To achieve optimum recognition rates, recognizers can limit themselves to a fixed character set. The Microsoft recognizer recognizes most of the characters in the ANSI character set and a standard set of gestures. The gestures are listed in online Help.

Any system recognizer for use within the U.S. should support at least a subset of the ANSI characters, including:

- One case (lower or upper) of the 26 letters
- Numbers
- A set of punctuation
- The standard gestures

There are no other requirements for the character set.

Multiple recognizers are supported. Any application using multiple recognizers must set the calling order of the recognizers. For example, a multilingual application could check to see if an item of input is an ANSI character and, if not, ask a Kanji recognizer if it could be a Kanji character.

It's also possible for an application to inform the recognizer that only a certain subset of the full character set is valid at a certain time. For example, the ZIPCODE field in the sample HFORM application uses the ALC_NUMERIC constant to specify a numeric-only

field, using the enabled alphabet field in the **RC** structure. The recognizer must not return a recognition result outside this range. This does not imply that a recognizer has to force a match; it can report that the character is unknown.

### Hot Spots

When a symbol is recognized, the recognizer may also identify a critical point (hot spot) on the gesture. For example, if the "X" symbol is used for deleting, the cross of the "X" points to the item to be deleted. If the recognizer identifies a hot spot for a recognized symbol, **lpsyg.rgpntHotSpot** is filled with the points. Different symbols will define different numbers of hot spots.

## Training

A recognizer can be trainable or nontrainable. A nontrainable recognizer has a fixed set of prototypes.

Trainable recognizers can be classified as passive trainers or active trainers. A passive trainer learns by automatically making modifications to its prototypes as the user is writing. It assumes that any uncorrected results are correctly identified. An active trainer forces the user to write in a training window and then positively verifies the input just written.

The Pen API supports both trainable and nontrainable recognizers. Passive training is internal to the recognition module; it is not constrained by the API.

## Symbol Values and Symbol Graphs

A recognizer communicates with an application by way of symbol values and symbol graphs.

### Symbol Values

Each glyph a recognizer can identify has an associated symbol value, which it returns to the application upon recognizing a glyph.

All system recognizers should also recognize a special set of glyphs (gestures) used as commands. The images for the gestures are defined in online Help.

The specific symbol values are discussed in more detail in Chapter 11, "Pen Messages and Constants."

### Symbol Graphs

Upon completion of recognition, a recognizer returns a symbol graph. 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.

Each element (**SYE**) of the symbol graph contains information about the recognized character: bounding rectangle, hot spots, and so on. Components of symbol graphs (**SYE**, **SYG**, and **SYC**) are listed in Chapter 10, "Pen Structures."

If a single entity recognized by the recognizer is mapped to a string of several symbol values, the recognizer creates multiple **SYE**s. This would be the case for recognizers that can recognize highly stylized runs of characters—for example, "ing"—in which the individual characters are not necessarily recognized.

Symbol graphs can be represented in different ways. Figure 7.2 shows one example.

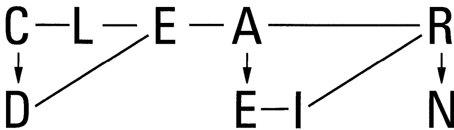



Figure 7.2. Symbol graph representation of handwriting

The symbol graph shown in Figure 7.2 represents these possible words:

| | |
|---|---|
| clear | cleeir |
| dear | deeir |
| clean | cleein |
| dean | deein |

This symbol graph could also be represented as:

```
{cl | d}e{a | ei}{r|n}
```

or

```
{clear | clean | cleeir | cleein | dear | dean | deeir | deein }
```

### Symbol Graph Grammar

A symbol graph is an array of symbol elements (**SYE**s). Each **SYE** has a symbol value.

The context-free grammar that defines the format of a symbol graph is listed in the following table. In place of **SYE**s, just the symbol values of the elements of the arrays are given. For **SYE**s that do not represent recognized symbols, such as SYV_NULL, the rest of the **SYE** structure is set to 0. Bold symbols in the grammar are the terminal symbols. The following symbols are meta symbols: ->, (, ), *, +, and ||.

| | | |
|---|---|---|
| Symbol graph | -> | **A* SYV_NULL** |
| A | -> | **SYV_BEGINOR** (B **SYV_OR**) * B **SYV_ENDOR** \|\| B |
| B | -> | One or more symbol values |

In practice, a recognizer will return a list of words (possibly concatenated with other lists, separated by white space symbol values), or it will return a word with a few alternatives for some of the characters.

The **EnumSymbols** function enumerates the possible paths through the symbol graph. It is up to the recognizer to construct the graph so that the interpretations of the symbol graph are enumerated in decreasing order of likeliness.

## The RC Structure

The core of the recognition process is the **RC** data structure.

There are a large number of parameters in the **RC** structure. In practice, however, an application has to deal with only a few of them. The application calls **InitRC** to set the default values and then adjusts certain parameters before making one of the **Recognize** calls.

The following table lists the **RC** structure fields. For more information, see Chapter 10, "Pen Structures."

| RC structure field | Description |
|---|---|
| **rc.alc** | Specifies the enabled alphabets for recognition (ALC_ values). |
| **rc.alcPriority** | Specifies the ALCs that have priority. |
| **rc.clErrorLevel** | Specifies the level at which the recognizer should reject input. |
| **rc.wTryDictionary** | Specifies the cutoff for enumerations per word. The minimum allowed is 1, the maximum 4096. The default value is 100. |
| **rc.dwAppParam** | Specifies the value defined by the application and passed on through the **RC** structure. |
| **rc.dwDictParam** | Specifies the value defined by the application and passed on to the dictionaries on the dictionary path. (This is for use on a per-dictionary basis.) |
| **rc.dwRecognizer** | Specifies the value defined by the application and passed on to the recognizers. (This is for use on a per-recognizer basis.) |
| **rc.guide** | Specifies the values to be used in setting guidelines on the screen for the recognizer. |
| **rc.hrec** | Specifies the handle of the recognizer to be used. |

| RC structure field | Description |
|---|---|
| **rc.hwnd** | Specifies the handle of the window to send the recognition results to. |
| **rc.lPcm** | Specifies the flags for ending recognition (the PCM_ values). |
| **rc.lpfnYield** | Specifies the callback function used by the recognizer before yielding. |
| **rc.lpLanguage [48]** | Specifies the list of language strings. |
| **rc.lpUser [32]** | Specifies the name of the current writer. If NULL, uses the standard prototype set. |
| **rc.lRcOptions** | Specifies recognition options (RCO_ values). |
| **rc.nInkWidth** | Specifies the thickness (0–15) of the ink during inking. |
| **rc.rectBound** | Specifies the bounding rectangle for inking. |
| **rc.rectExclude** | Specifies the area where a pen down event will end recognition. |
| **rc.rgbfAlc [32]** | Specifies the bit field used for enabled characters. |
| **rc.rgbInk** | Specifies the color (nearest solid color) to use for inking. |
| **rc.rglpdf [MAXDICTIONARIES]** | Specifies dictionaries called by the recognizer to convert symbol graphs to strings. If the **rglpdf[0]** field is NULL, the NULL dictionary is used. The path should be NULL-terminated. |
| **rc.rgwReserved** | Used internally; applications should not modify these values. |
| **rc.wEventRef** | Specifies the event that begins the recognition process. |
| **rc.wRcDirect** | Specifies the direction of writing (RCD_ values). |
| **rc.wRcOrient** | Specifies the orientation of the tablet (RCOR_ values). |
| **rc.wRcPreferences** | Specifies user preferences—for example, writing hand (left or right) or whether gestures are positional (RCP_ values). |

| RC structure field | Description |
|---|---|
| **rc.wResultMode** | Specifies the timing of the results messages to be sent back to the specified window (RRM_ values). |
| **rc.wTimeOut** | Specifies the time-out period (milliseconds). |

# How a Custom Recognizer Interacts with the RC Manager

This section describes the interface from the RC Manager to the particular recognizer installed. It also summarizes the entry points the OEM recognizer can call.

## Custom Recognizer Functions

The recognition module is written as a Windows DLL. The standard DLL initialization and termination entries (**LibMain** and **WEP**) are called.

These functions must be supplied in an OEM recognizer DLL:

| Function | Description |
|---|---|
| **CloseRecognizer** | Called when the recognizer is uninstalled by an application |
| **ConfigRecognizer** | Sets recognizer-specific parameters |
| **InitRecognizer** | Gives the recognizer a chance to do any initialization before receiving the first request for recognition |
| **RecognizeDataInternal** | Begins sampling pen data stored in the buffer and converts the input to recognized symbols |
| **RecognizeInternal** | Begins sampling pen data from the tablet and converts the input to recognized symbols |
| **TrainContextInternal** | Performs training using contextual (recognition) information |
| **TrainInkInternal** | Performs context-free training |

## Division of Responsibility Between Recognizer and RC Manager

The recognizer is responsible for mapping raw data in the form of pen input to the symbols the input represents. The RC Manager provides support both before and after this processing. The RC Manager sets up the pen driver for data collection and display. After the recognizer identifies results, the RC Manager performs the necessary dictionary processing and fills in the remaining fields.

The following outline of pseudocode provides more details on this division of work:

```
Recognize( )
  Validate RC values and replace any requests for default values
  with the actual values.
```

```
Prepare for inking and data collection
SetCapture(rc.hwnd)
Hide Cursor if necessary
Remove any existing mouse messages
ReleaseCapture()

Set conditions for ending recognition

Begin Actual recognition work
error code = RecognizeInternal()
Restore cursor
return error code.
```

---

Custom recognizer

A general recognizer has the following general form:

```
{
Allocate memory to buffer results and raw data;

while (GetPenHwData(..) == REC_OK)
  {
  Yield sometimes;
  Add points to pendata buffer
  if (overflow)
    {
    FEndPenCollectionMode(REC_OOM);
    continue;
    }
  if (enough data to recognize)
    {
    Perform recognition;

    Fill RCRESULT struct;
    Call lpFuncResults(..rcresult..);
    if (lpFuncResults == 0)
      return (valid value for RecRecognize);
    }
  }

if (still some raw data left)
  {
  Perform recognition;
  Fill RCRESULT struct;
  Call lpFuncResults(..rcresult..);
```

```
    }
  Free buffer memory;

  return (valid value for Recognize);
  }
```

# A Sample Recognizer

The SREC.C file, located in the PENSDK\SAMPLES\SREC directory, is a skeleton program containing all of the necessary functionality for a simple recognizer. The recognizer is a DLL loaded at runtime by the pen application. This particular recognizer is used by the PENAPP sample application.

This recognizer accepts only a single stroke of data points. The recognizer takes the beginning and ending points of the stroke and calculates the nearest compass direction of the line formed by these endpoints.

The recognizer then fills out the symbol graph (passed through the *lParam* parameter of the application's window procedure on the WM_RCRESULT message) using the following special codes.

| Value | Direction |
|---|---|
| syvEast | Right |
| syvSouth | Down |
| syvWest | Left |
| syvNorth | Up |
| syvDot | Single tap |

The PENAPP application that uses this recognizer sets the following **RC** structure item:

```
lPcm = PCM_PENUP | any other flags.
```

The PCM_PENUP flag guarantees one single stroke and then recognition.

## Windows Dynamic-Link Library Functions

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 initializations and unlocks the data segment of the library. **WEP** is the standard DLL termination function. **LibEntry**, in the file LIBENTRY.ASM, is identical to the source used in the sample DLL in the SDK for MS Windows version 3.1.

## Stub Functions

A large number of functions are required to be present in any recognizer even if they do little, if any, processing. The following functions do little actual processing but are stubs for your enhancements. In general, they return some value to the calling program. The values returned are those for the sample recognizer discussion here.

| Stub function | Description | Returns |
|---|---|---|
| **InitRecognizer** | Initializes recognizer and loads any necessary data | TRUE |
| **CloseRecognizer** | Closes recognizer and saves any necessary data | VOID |
| **ConfigRecognizer** | Configures the recognizer for special options | TRUE |
| **TrainInkInternal** | Performs context-free training | FALSE |
| **TrainContext** | Performs training using contextual (recognition) information | FALSE |

## ConfigRecognizer

The **ConfigRecognizer** function configures the recognizer for special options. Typically, it is called by the application or system Control Panel to set recognizer-specific options or to query its capabilities.

Different recognizers will allow for different levels of configuration. For example, a recognizer can provide a configuration dialog to enable or disable cursive input. This sample recognizer has no configuration options. The configuration dialog is called from the Recognizer Dialog of the Control Panel.

In the following code fragment, the SREC recognizer returns only the identification string of the recognizer. Also, note that the SREC recognizer cannot be set as the system recognizer. Since SREC does not support standard editing gestures or characters, it is not a valid system default recognizer.

```
WORD FAR PASCAL ConfigRecognizer(WORD wConfig, WORD wParam, LONG lParam)
  {
  WORDwRet = TRUE;

  wParam;    // to prevent CS 5.1 compiler warning message

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

    case WCR_QUERY:
      wRet = FALSE;    /* Does not support configuration */
      break;

    case WCR_CONFIGDIALOG:
      break;

    case WCR_DEFAULT:
```

```
      wRet = FALSE;    /* Incapable of being system default */
      break;

    case WCR_RCCHANGE: /* Change any internal parameters based on */
                         /* new information - for example current user */
      break;

    case WCR_VERSION:
      wRet = 0x0103;   /* Recognizer version 3.1 */
      break;

    case WCR_TRAIN:
      wRet = TRAIN_NONE;     /* Does not support training */
      break;

    case WCR_TRAINSAVE:
      wRet = FALSE;
      break;
    }
  return wRet;
  }
```

## Recognition Functions

The two main components of any recognizer DLL are the **RecognizeInternal** and **RecognizeDataInternal** functions. They handle the recognition from hot and cold recognition, respectively.

**RecognizeInternal** is the custom OEM recognizer function. It receives pen stroke input through the **GetPenHwData** function and performs recognition. The recognition results are passed back to the RC Manager in an **RCRESULT** structure through the **lpFuncResults** function.

In the sample program, you create the symbol graph in the **CalcNearestDir** function, which is called from the private function **DoRecognition**. In this sample, no recognition is performed concurrently with data collection. As a result, **RecognizeInternal** just collects ink and passes it on to **RecognizeDataInternal**.

### RecognizeInternal

For the SREC recognizer, this function receives input of data points of one stroke and calculates the closest compass direction of the stroke. Unlike a typical recognizer, this recognizer collects all data points first before performing any calculation.

The function begins by allocating the OEM data buffer. Next, it runs the data collection loop using **GetPenHwData** to collect pen points. The **lpfnYield** field of the **RC** structure controls the yielding of pen tasks to other waiting Windows tasks. Finally, the function calls **DoRecognition** to continue the recognition process.

The **DoRecognition** function contains the common code between **RecognizeInternal** and **RecognizeDataInternal**. **DoRecognition** also fills in the **RCRESULT** structure.

For the SREC sample program, the **RecognizeInternal** function receives input of data points of one stroke and calculates the closest compass direction of the stroke.

The recognizer assumes the PCM_PENUP is set, so it does no time-out checking.

```
REC FAR PASCAL RecognizeInternal(LPRC lprc, LPFUNCRESULTS lpFuncResults)
  {
  WORDrgpntOem[cpntMax*MAXOEMDATAWORDS];     /* temporary buffer */
  POINT rgpnt[cpntMax];                     /* actual data buffer */
  WORDcYield= 0;     /* yield count */
  BOOLfSaveAll= (lprc->lRcOptions & RCO_SAVEALLDATA) != 0;
  LPVOID  lpvOem = (LPVOID)(fSaveAll ? rgpntOem : (LPVOID) NULL);
  REC    rec = REC_OK;
  HPENDATA hpendata;
  HPENDATA hpendataT = NULL;
  RCRESULT rcresult;
  STROKEINFO si;

  /* Allocate OEM data buffer */

  if ((lprc->lPcm & PCM_PENUP) == 0)
    return REC_NOPENUP;    // Recognizer-specific error

  if ((hpendata = CreatePenData(NULL, (fSaveAll ? -1 : 0),
PDTS_STANDARDSCALE, GMEM_SHARE)) == NULL)
    return REC_OOM;

  /* Data input loop */

  while ((rec = GetPenHwData(rgpnt, lpvOem, cpntMax, 0, &si)) == REC_OK)
    {
    if (si.cPnt != 0)
      {
      if ((hpendataT = AddPointsPenData(hpendata, rgpnt, lpvOem, &si)) ==
NULL)
        {
        rec = REC_OOM;
        break;
        }
      hpendata = hpendataT;
      }

    if (cYield++ % 5)
      {
      (*lprc->lpfnYield)();    // Yield
      }
    }
```

```
/* Copy last point.  Note that only the last point really counts
   if rec == REC_TERMPENUP */

if (hpendataT != NULL && rec == REC_TERMPENUP)
  {
  /* Normal ending of preceding loop.  Add the pen up stroke */
  if ((hpendata = AddPointsPenData(hpendata, rgpnt, lpvOem, &si)) == NULL)
    rec = REC_OOM;
  }


if (rec == REC_TERMPENUP)
  {
  // Send results back to the application.
  DoRecognition(lprc, hpendata, &rcresult);
  (*lpFuncResults)((LPRCRESULT) &rcresult, rec);
  }

// Free up memory used to save data.  If you want the application to
// save it, it must make a copy or set the RCO_SAVEHPENDATA flag.

if (hpendata != NULL && (lprc->lRcOptions&RCO_SAVEHPENDATA)==0)
  DestroyPenData(hpendata);

return rec;
}
```

### CalcNearestDir

The user-defined function, **CalcNearestDir**, is called with the endpoints. This function calculates the closest compass direction of the line defined by the two end points. The **CalcNearestDir** function concludes by setting the confidence level of recognition.

```
VOID NEAR PASCAL CalcNearestDirection(LPPOINT lppointEnds)
  {
  int  dx;
  int  dy;
  BOOL fIsEastward;
  BOOL fIsSouthward;
  BOOL fIsHoriz;

  extern SYE syeGlobal;

  dx = (lppointEnds+1)->x - lppointEnds->x;
  dy = (lppointEnds+1)->y - lppointEnds->y;

  fIsEastward = dx > 0 ? TRUE : FALSE;
  fIsSouthward = dy > 0 ? TRUE : FALSE;
  fIsHoriz = ABS(dx) > ABS(dy) ? TRUE : FALSE;
```

```
if (fIsHoriz)
  {
  syeGlobal.syv = fIsEastward ? syvEast : syvWest;
  }
else
  {
  syeGlobal.syv = fIsSouthward ? syvSouth : syvNorth;
  }
if (dx == 0 && dy == 0)
  {
  syeGlobal.syv = syvDot;
  }

syeGlobal.cl = 100;       /* Set confidence level */
syeGlobal.lRecogVal = 0L;  /* Not used */
}
```

# Dictionaries

The dictionary component, like the recognizer, is a replaceable component. Although the use of a dictionary is optional, it may increase the recognition accuracy of an application.

A dictionary is implemented as a dynamic-link library (DLL) with only one exported function. This function, **DictionaryProc**, is called with different subfunction arguments in order to perform some action. For a complete description of the **DictionaryProc** function, see Chapter 9, "Pen API Reference."

The application tells the system to call the dictionary before returning the WM_RCRESULT message. Alternatively, the application can call the dictionary itself to perform a postrecognition search.

Dictionaries can perform a number of different services, of which the most common service is to provide a word list used to find exact matches. A dictionary can also perform spell checking, format checking, or macro expansion of a string.

Of all the services a dictionary can perform, the most common is for simple word matching. Included with this SDK are two dictionary DLL files, MAINDICT.DLL and USERDICT.DLL. The MAINDICT.DLL file is a general-purpose language dictionary, providing, among other things, a facility to find words in any of several European languages. The USERDICT.DLL file is a dictionary DLL and can be used for word lookup using application-supplied word lists.

The EXPENSE sample included with this SDK shows how a simple dictionary DLL can be used to provide a custom word list for an application and to perform prefix matching. This example is discussed in more detail later in this chapter.

## The RC Structure and Dictionary Processing

The simplest implementation of a dictionary is to let the system control when the dictionary is called. When an application sets the **RC** structure to be used during

recognition, it doesn't need to change any of the default values to get the system to call the default system dictionary. The fields in the **RC** structure that are of interest to the dictionary are **rc.lRcOptions, rc.rglpdf, rc.wTryDictionary** and **rc.dwDictParam.** An application uses these fields to fine-tune how the system will respond when the dictionary is called.

The core of the information needed for the dictionary DLL is in the **RC** data structure. Although there are a large number of parameters in the **RC** structure, in practice, a dictionary application will have to deal with only a few of them.

The following sections list the **RC** structure fields that are used by the dictionary or by an APPLICATION using a dictionary DLL. For more information, see Chapter 10, "Pen Structures."

## rc.lRcOptions

This variable is used by both the recognizer and the dictionary. The RCO_ constants that pertain to the dictionary are RCO_NOSPACEBREAK and RCO_SUGGEST. The application should use these constants in the following way.

RCO_NOSPACEBREAK specifies that when the system calls the dictionary, it should not preprocess the information with regard to space breaks. If the user writes "hi there" and the RCO_NOSPACEBREAK flag is set, the dictionary should receive the string "hi there." If the flag is not set, the dictionary should receive two separate strings. The system asks the dictionary first to identify the string "hi" and then to process "there." RCO_NOSPACEBREAK should be used only in cases where white space is a significant part of the information. Usage of this flag in language strings processing could significantly slow the lookup.

If the constant RCO_SUGGEST is set, the system will call the dictionary with the DIRQ_SUGGEST message—but only under special circumstances. DIRQ_SUGGEST is sent to the dictionary only if the RCO_SUGGEST option is set in the **RC** structure and the dictionary has failed to return success when processing the DIRQ_STRING messages.

The DIRQ_SUGGEST message is designed to be sent to the dictionary after the DIRQ_STRING messages have been processed. When the dictionary processes the DIRQ_STRING messages, it looks for the match in the word list that fits the enumeration it has received. The DIRQ_SUGGEST message is designed for spell checking or macro expansion of the enumeration.

Say, for instance, that the user writes "SDK," but the word "SDK" is not in the dictionary. If the RCO_SUGGEST flag is set, the dictionary will be called with the best enumeration of the string, and it will be able to return its best guess. In this case, the dictionary might return "Software Development Kit" instead of "SDK." If the RCO_SUGGEST flag is not set in the **RC** structure, this message will never be sent to the dictionary.

## rc.rglpdf

Within the PENWIN.H header file, **rc.rglpdf** is an array of far addresses to dictionary functions. Currently, the system supports a maximum of 16 dictionaries, as specified in the constant MAXDICTIONARIES. The number and order of these callback functions is important.

If there are no dictionaries in the list, the system uses the NULL dictionary to extract the symbols from the symbol graph with the highest confidence level, and it returns this with the WM_RCRESULT message.

If there are one or more dictionaries, the system calls the **DictionarySearch** function, which uses DIRQ_SYMBOLGRAPH to call all the dictionaries. If none of the dictionaries respond with a match, the **DictionarySearch** function enumerates the symbol graph and calls each dictionary function with DIRQ_STRING for each enumeration.

Suppose, for example, the symbol graph returned from the recognizer is:

```
{1|l}ow
```

The dictionary will be called with each enumeration of the symbol graph—"1ow" (the numeral "1" plus "ow") followed by "low" (with the letter "l" as the first character). With each call, the dictionary has the option of either accepting or rejecting the input. If the input is not accepted, the system will continue to send enumerations to the dictionary until there are no more enumerations left. If the dictionary accepts the input, the system immediately sends the ending message DIRQ_STRING with *lpIn* set to NULL.

After all of the enumerations have been sent, the system calls the dictionary with a DIRQ_STRING message with the *lpIn* parameter set to NULL. This informs the dictionary that there are no more enumerations left. If there is only one dictionary in the search list, the system sends the DIRQ_SUGGEST message.

If there are multiple dictionaries in the list, the system determines the ORDER in which they are called by their positions in the list and the confidence of the enumeration. The system starts by getting the first enumeration of the symbol graph. For every enumeration, it calls the dictionaries, starting with the one located at position zero, with the DIRQ_STRING message. If the first dictionary in the list rejects the input, the second dictionary attempts the same enumeration.

This process continues until there are no more enumerations or until a dictionary returns success. If, at any time, one of the dictionaries reports a match, the system sends a DIRQ_STRING message to every dictionary with *lpIn* set to NULL, indicating that the search is over. If no match is found, the system may then call the dictionaries with the DIRQ_SUGGEST message in the same order. Again, the DIRQ_SUGGEST message will be called only if the RCO_SUGGEST flag is set.

## rc.wTryDictionary.

This variable should be set with the "cutoff" threshold value to be used by the system when determining which enumeration strings from the symbol graph to send to the dictionary. The **wTryDictionary** field of the **RC** structure specifies the cutoff for enumerations per symbol graph. The minimum number allowed is 1 and the maximum is 4096. The default value is 100.

## rc.dwDictParam

This variable is reserved for dictionaries that require more specific information when called by the system. The information placed in this variable is passed on to the dictionary

function as the ID parameter. Applications can use this variable to store information that might be specific to the **RC** structure the application is currently using.

When using **rc.dwDictParam**, it is important that the application controls which dictionary is called, because the **dwDictParam** is sent to every dictionary in the list. For this reason, if this variable is used, the application should be aware of all dictionaries in the list so that only the dictionaries that can support this ID value will receive the message.

Most dictionaries, including the ones included with the SDK, will ignore this parameter.

Placing the dictionary function callback address in the **rc.rglpdf** list is not the only way to get a dictionary to work. Every WM_RCRESULT message contains the symbol graph that represents the current user's input. To simulate the operations of the system, you can call the **DictionarySearch** function, and it will mimic the calls the system makes before your application receives the WM_RCRESULT message. This can be useful for calling a dictionary with information that needs to be recognized in context.

For example, if the user writes "at" on top of "is" in the word "this," you might want to have the application defer the dictionary search until after it receives the WM_RCRESULT message. The application would calls the **DictionarySearch** function with the symbol graph representing "that," instead of having the system call the dictionary with "at."

## Subfunction Messages Used in a Dictionary DLL

When designing a dictionary, you need to be aware of what messages you'll need to support for system services and which messages you might want to support for enhanced functionality. The most important message is the DIRQ_STRING message. All other messages are optional.

The messages defined by Microsoft can be combined into logical groups. There are basically three types of messages: system messages, dynamic manipulation messages, and other messages. The following table lists the messages associated with each group.

| Dictionary message group | Messages |
|---|---|
| System messages | DIRQ_STRING, DIRQ_RCCHANGE, DIRQ_SUGGEST, DIRQ_SYMBOLGRAPH, DIRQ_INIT, and DIRQ_CLEANUP |
| Dynamic manipulation messages | DIRQ_ADD, DIRQ_CLOSE, DIRQ_DELETE, DIRQ_FLUSH, DIRQ_OPEN, and DIRQ_SETWORDLISTS |
| Other messages | DIRQ_CONFIGURE, DIRQ_DESCRIPTION, DIRQ_QUERY, DIRQ_USER, and DIRQ_COPYRIGHT |

The dictionary can support any number of these messages. However, a dictionary must support DIRQ_OPEN, DIRQ_CLOSE, DIRQ_QUERY, DIRQ_STRING, DIRQ_DESCRIPTION, DIRQ_INIT, DIRQ_CLEANUP, and DIRQ_COPYRIGHT.

For detailed information about the messages, see Chapter 11, "Pen Messages and Constants."

## A Sample Dictionary

The CUSTDICT.C file (located in the \PENSDK\SAMPLES\EXPENSE directory) is a skeleton program containing all the necessary functionality for a simple dictionary. The dictionary is a DLL loaded at runtime by the pen application. This particular dictionary is used by the EXPENSE sample application (also located in the same directory) to perform dictionary requests such as DIRQ_QUERY and DIRQ_STRING.

The EXPENSE.C sample is a straightforward pen application. It contains several edit fields typical of a generic expense report (Name, Employee #, Dept. Items, and so on). The application registers itself as a pen-aware application so that the edit controls are replaced by hedit controls when operating on a pen computer.

In the expense report, the user dictionary is used in processing results in the Name and Dept. Name fields, and the custom dictionary is used in the Expense Item field. There are separate word lists associated with both the Name and Dept. Name fields. These word lists are specified in the PENWIN.INI file in the following form:

```
[Expense]
namedict=<full path to the names dictionary>
deptnamedict=<full path to the department name dictionary>
```

NAMES.DIC (a Names word list) and DEPTNAME.DIC (a Department Name word list) have been included with the sample application. If there are no entries in the PENWIN.INI file, these default word lists are checked for in the working directory.

The custom dictionary maintains an internal array of words instead of having a word list file. This array contains the following words: Taxi, Food, Flight, Hotel, Misc.

If an application or the RC Manager needs the custom dictionary to perform some predefined task, a call is made to **DictionaryProc**. The following table lists the requests that the custom dictionary has implemented. These are listed as **case** statements in the **DictionaryProc** function.

| DIRQ_ sub function | Description |
|---|---|
| DIRQ_QUERY | Allows the calling program to ask if a specific DIRQ request is supported. |
| DIRQ_DESCRIPTION | Returns, in one of the calling parameters, the name of the DLL in which the custom dictionary exists. |
| DIRQ_STRING | Determines if the supplied symbol list contains a word that is accepted by the dictionary. FALSE is returned if the word is not accepted. If the word is accepted, the custom dictionary places a symbol list representation of that word in one of the calling parameters and returns the number of symbols in the list. |

| DIRQ_ sub function | Description |
|---|---|
| DIRQ_SUGGEST | If the dictionary receives this request, it is usually from the RC Manager after all enumerations of the symbol graph have been processed with DIRQ_STRING requests. It is up to the custom dictionary to determine what kind of processing will take place at this point.<br>In the CUSTDICT sample, a simple, non-case-sensitive prefix matching is performed on the word in the supplied symbol list. The **FBestGuess** function in CUSTDICT is used for the prefix search. Therefore, if the user writes in "Fl," the custom dictionary returns "Flight" in its suggested symbol list. However, if a prefix is not matched, the custom dictionary returns a NULL symbol list, which signals to the caller that the lookup was unsuccessful. |

Many types of functionality can be added to the custom dictionary. For example, in the Date field on the expense report, date verification could be performed. Notice that the same dictionary can be used for different functions based on the **dwDictParam** of the **RC** structure. The **dwDictParam** is passed to the dictionary and can be used to signal the field that is being written in.

## Loading and Initializing the Dictionary DLL

A custom dictionary is always implemented as a Windows dynamic-link library. In addition to the necessary functions of a DLL such as **LibMain** and **WEP**, the **DictionaryProc** function must also be implemented in the DLL.

The first two functions in the CUSTDICT recognizer are the standard Windows functions required in any DLL—**LibMain** and **WEP**. **LibMain,** the main DLL function, is analogous to **WinMain**; it performs any needed initializations and unlocks the data segment of the library. **WEP** is the standard DLL termination function. **LibEntry**, in the file LIBENTRY.ASM, is identical to the source used in the sample DLL.

During the initialization of the instance of the application in the **FInitInstance** function in EXPENSE.C, the user dictionary and the custom dictionary are loaded with calls to **LoadLibrary**.

Once the libraries are loaded and the addresses of the dictionary procedures have been determined, the word lists associated with the user dictionary are loaded. This is done by calling the dictionary procedures with the DIRQ_OPEN message and specifying the path to the word list file. This call returns a handle to the word list in one of the calling parameters, which can be used when the word list is set.

```
BOOL NEAR PASCAL FInitInstance(HANDLE hInstance, HANDLE hPrevInstance,
                  int cmdShow)
  {
  .
  . Code omitted for clarity
  .
  /* load the library containing the user dictionary procedure */
  if ((hUserDictLib = LoadLibrary(szUserDictLib)) != NULL)
    {
    if ((lpdfUserDictProc = GetProcAddress(hUserDictLib, szUserDictProc))
              == NULL)
    return FALSE;
    }
  else
    return FALSE;

  /* load the library containing the custom dictionary procedure */
  if ((hCustomDictLib = LoadLibrary(szCustomDictLib)) != NULL)
    {
    if ((lpdfCustomDictProc = GetProcAddress(hCustomDictLib,
        szCustomDictProc)) == NULL)
    return FALSE;
    }
  else
    return FALSE;

  /* Open all the user dictionary word lists */
  for (i = 0; i < SIZE_WORDLIST; i++)
    {
    if (GetPrivateProfileString((LPSTR)szAppName,
        (LPSTR)rgwordlist[i].szProfileString, (LPSTR)rgwordlist[i].szDefault,
        szWordListPath, sizeof(szWordListPath), (LPSTR)szIniFile))
      {
      (*lpdfUserDictProc)(DIRQ_OPEN, szWordListPath,
          &rgwordlist[i].iList, NULL, NULL,NULL);
      }
    }
```

## Calling the Dictionary DLL

**ExpenseDictionaryProc**, a subclassed user dictionary **DictionaryProc** function, has been set up to determine when to set a word list in the user dictionary. **ExpenseDictionaryProc** has the same parameters as **DictionaryProc**. Once the word list has been checked, the user dictionary is called with the same parameters **ExpenseDictionaryProc** received. This passes the initial dictionary request to the user dictionary.

In the process of creating the report, the **RC** structure associated with each edit field is retrieved using the WM_HEDITCTL and HE_GETRC messages. Instead of entering the user dictionary in the **RC** structure, **ExpenseDictionaryProc** is specified. The **dwDictParam** is set to the corresponding index into the edit field array. This parameter signals to the **ExpenseDictionaryProc** which edit field is active. Once the **RC** elements have been updated accordingly, the **RC** structure is set in the edit control using the WM_HEDITCTL and HE_SETRC messages.

**ExpenseDictionaryProc** uses a global variable to check if the active edit field differs from the edit field that last accessed the user dictionary. If the edit field is different, the word list associated with that field is set by calling the user dictionary procedure directly with the DIRQ_SETWORDLIST request.

Any recognition results that occur in the Names and Department Name fields will eventually be processed by the user dictionary using the set wordlist. If no matches are found, the user dictionary is not called with a DIRQ_SUGGEST message, because the RCO_SUGGEST flag has not been set; therefore, the RC manager returns the best guess as the result.

The expense report's use of the user dictionary demonstrates the ability to use the same dictionary with different word lists for different edit fields.

After recognition occurs in the Expense Item field of the expense report, the custom dictionary is called by the RC Manager with the results from the recognizer and the DIRQ_STRING request. If the custom dictionary finds a match between the supplied enumeration of the symbol graph and internal word list, the matched text appears in the Expense Item field. If no matches are found, the RC Manager calls the custom dictionary again with a DIRQ_SUGGEST request. The custom dictionary then performs prefix matching on the specified symbol list.

# Pen API Overview

Microsoft Windows for Pen Computing is an extension to Microsoft Windows version 3.1. It builds upon the Windows application programming interface (API) to provide a rich environment for developing pen-based applications.

Several components in Microsoft Windows for Pen Computing are completely replaceable by third-party components. Both the dictionary and the recognition components are replaceable. The term dictionary refers to any postrecognition processing of the handwriting. The term recognizer refers to the component that performs the mapping of raw input to recognized symbols. (Recognizers are supplied by various software and hardware vendors, including Microsoft.) A third-party recognizer can be designed to recognize Kanji, Arabic, or even shorthand. For an explanation of how a simple recognizer and dictionary operate, see Chapter 7, "Replaceable Components: Recognizers and Dictionaries."

Device drivers can also be replaced by third-party components. To simplify the creation of applications, hardware-specific code has been isolated in the pen driver and the display-driver components. For the average application programmer, there is no need to modify the standard device drivers. For the OEM and ISV programmer, sample source code for both a tablet and a display driver are provided with this SDK.

Figure 8.1 shows the basic structure of the Pen components. The shadowed modules are replaceable.

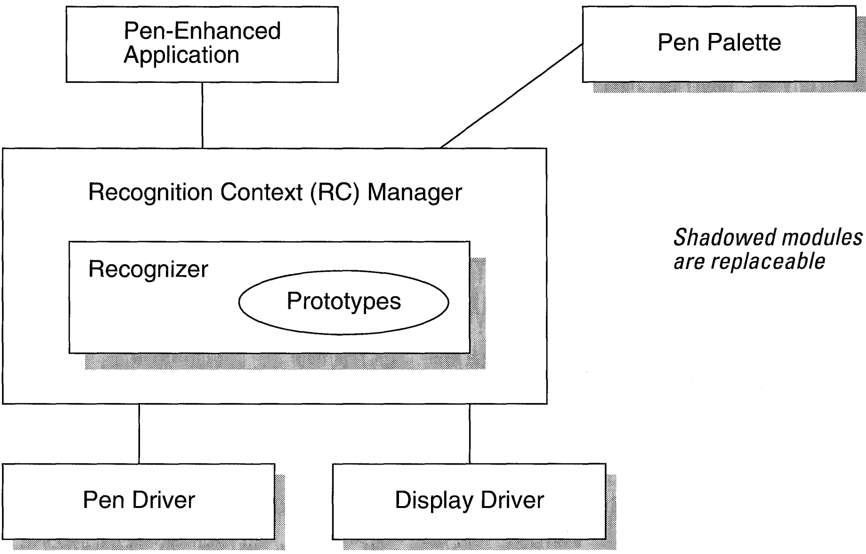



Figure 8.1. Pen components

The following table describes the Pen components.

| Component | Description |
|---|---|
| Pen-enhanced application | An application that makes use of the RC Manager to enable the use of a pen by the application. |
| Pen Palette | An application that mediates communication between the pen and an existing, unmodified Windows application. An OEM may provide enhanced capabilities in a Pen Palette application. |
| Display driver | The component that provides the ability to ink during recognition. *Inking* shows pen activity on the video screen or tablet surface. |
| Pen driver | The component that manages input from the digitizer tablet and provides mouse emulation. |
| Prototypes | An internal database used by the recognizer. Prototypes are compared with user input in the recognition process. |
| RC Manager | Used to denote functions, code, and functionality that Microsoft has added to Windows to support handwriting input and stylus devices. |
| Recognizer | The module that performs recognition. It compares user input to a set of prototypes. |

The flow of data from pen to RC Manager to application is discussed in Chapter 5, "A Sample Pen Application."

# Pen API Categories

The Pen API is an extension to the standard Microsoft Windows API. The Pen API contains the functions, messages, data structures, data types, statements, and files you need to create programs and device drivers that run with Microsoft Windows for Pen Computing.

There are six categories of API functions. The following table briefly describes each group. Each of these categories is discussed in more detail following this table.

| API Category | Description |
|---|---|
| Pen interface | Adds pen functionality to applications. If you use the Microsoft-supplied recognizer and device driver, this may be the only group of functions you'l need. |
| Pen data | Manipulates the pen data memory block (accessed with the **HPENDATA** structure), the primary mechanism for passing the information captured from the pen between an application and other applications and recognizers. |
| Custom recognizer | Functions to be provided by the recognizer component. The application does not call these functions directly. |
| Pen module | Functions called by pen drivers and recognizers to manipulate data coming from the pen driver. |
| Pen driver | Functions provided by pen drivers. |
| Display driver (inking) | Functions provided by display drivers. |
| Dictionary | Functions provided by a custom dictionary. |

## Pen Interface Functions

There are three methods for creating a pen application. One method is to use the **ProcessWriting** function, which simplifies the process of collecting handwritten input. For information about this method, see Chapter 6, "Using Pen Controls and the **ProcessWriting** Function."

The following paragraphs discuss the two remaining methods, using a recognizer within a window and using the Pen controls.

## To Use a Recognizer within a window

This method uses a recognizer within a window to generate handwriting entry and recognition behavior. The RC Manager invokes the recognizer for the application. The procedure is as follows:

1. Install a custom recognizer (if appropriate), using **InstallRecognizer.**
2. When pen input begins, initialize the **RC** data structure.
3. Call the recognizer.
4. Receive the recognized data in a Windows message.
5. Process the data with your program.
6. When finished, unload the custom recognizer (if appropriate).

For an example that uses this procedure, see the discussion of \PENSDK\SAMPLES\SREC in Chapter 5, "A Sample Pen Application."

## To Use the Pen Controls

The hedit control completely replaces the default edit class of Windows. In addition to the normal edit control functionality, hedit supports direct handwriting input. That is, in an hedit control, you can mix keyboard, mouse, and pen input.

The bedit control is an entirely new edit class that implements boxed input. The bedit control also accepts keyboard and mouse input.

For a sample program that uses hedit and bedit controls, see the discussion of \PENSDK\SAMPLES\HFORM in Chapter 6, "Using Pen Controls and the **ProcessWriting** Function."

## Function Subcategories

The Pen Interface functions install and call recognizers, create handwriting edit controls, and unload recognizers when completed.

There are five subcategories of functions in the Pen Interface category:

- Virtual Event Layer
- Recognition
- Symbol Manipulation
- Training
- Utility

### Virtual Event Layer

The Virtual Event Layer subcategory provides the mechanism for sending virtual mouse or keyboard events to Windows. The following table describes the functions in this subcategory.

| Function | Description |
|---|---|
| **AtomicVirtualEvent** | Blocks out physical pen events while posting virtual events |
| **PostVirtualKeyEvent** | Sends a virtual key code event to Windows |
| **PostVirtualMouseEvent** | Sends a virtual mouse event to Windows |

### Recognition

The Recognition subcategory provides the mechanism for initializing, installing, and unloading the recognizer; for recognizing data; and for correcting errors. The following table describes the functions in this subcategory.

| Function | Description |
|---|---|
| **CorrectWriting** | Displays Edit Text Window. |
| **EmulatePen** | Emulates text I-beam. In general, you should use **ProcessWriting** or hedit controls instead. |
| **ExecuteGesture** | Converts a gesture to a set of keystrokes that the user has mapped. |
| **InitRC** | Initializes recognition context for the recognizer. |
| **InstallRecognizer** | Loads a specified recognizer. |
| **IsPenAware** | Checks application's capability of handling pen events. |
| **ProcessWriting** | Runs high-level recognition services. |
| **Recognize** | Begins sampling pen data from the tablet and converts the input to recognized symbols. |
| **RecognizeData** | Converts pen data stored in a buffer to recognized symbols. |
| **RegisterPenApp** | Modifies the behavior of an edit control within an application to behave as an hedit control. |
| **SetRecogHook** | Installs or removes a recognition hook. |
| **ShowKeyboard** | Displays or hides the on-screen keyboard. |
| **UninstallRecognizer** | Unloads a specified recognizer. |

### Symbol Manipulation

The Symbol Manipulation subcategory provides the mechanism for converting symbols to characters.

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 a null-terminated array of symbols. A symbol graph is a compact representation of the alternatives recognized.

The following table describes the functions in this subcategory.

| Function | Description |
|---|---|
| **CharacterToSymbol** | Converts an ANSI string to an array of symbol values |
| **GetSymbolCount** | Returns the number of possible symbol strings that can be generated from the symbol graph |
| **EnumSymbols** | Enumerates strings of symbols from a symbol graph |
| **FirstSymbolFromGraph** | Returns the array of symbols that is the most likely interpretation of a specific symbol graph |
| **GetSymbolMaxLength** | Gets the length of the longest symbol string generated from the symbol graph |
| **SymbolToCharacter** | Converts an array of symbols to an ANSI string |

### Training

The Training subcategory provides the mechanism for training the ink to improve recognition. The following table describes the functions in this subcategory.

| Function | Description |
|---|---|
| **TrainContext** | Informs the recognizer that the raw data input represents the symbol value results |
| **TrainInk** | Informs the recognizer that the raw data input represents the symbol value results |

### Utility

The Utility subcategory provides a variety of functions for general maintenance, coordinate conversion, and low-level functions. The following table describes the functions in this subcategory.

| Function | Description |
|---|---|
| **BoundingRectFromPoints** | Returns the bounding rectangle of an array of points. |
| **DPtoTP** | Converts display coordinates to tablet coordinates. |
| **GetGlobalRC** | Gets the current default settings for the specified recognition context. Normally, an application gets this information at **RC** initialization time. However, if you need to query these values at some other time, use **GetGlobalRC**. |
| **GetMessageExtraInfo** | Windows function used to extract extra information passed along with mouse event messages. |
| **GetPenAsyncState** | Gets the barrel button state of the pen. |
| **GetVersionPenWin** | Gets the version number. |

| Function | Description |
|---|---|
| **SetGlobalRC** | Sets default settings for the specified recognition context. This function should be called only from the pen Control Panel program. |
| **SetPenHook** | Captures low-level pen events. |
| **TPtoDP** | Converts tablet coordinates to display coordinates. |

## Pen Data Functions

The HPENDATA handle to the pen data memory block is the primary mechanism for passing the information captured from the pen between an application and other applications and recognizers.

The pen data functions manipulate the pen data memory block. The following table describes the functions in this category.

| Function | Description |
|---|---|
| **AddPointsPenData** | Adds an array of new points and OEM data to an existing pen data memory block |
| **BeginEnumStrokes** | Initializes the pen data memory block |
| **CompactPenData** | Performs various memory-saving operations on the pen data memory block |
| **CreatePenData** | Allocates memory for a new pen data memory block and initializes its header |
| **DestroyPenData** | Frees up memory associated with a pen data memory block |
| **DrawPenData** | Displays the ink in an **hDC** |
| **DuplicatePenData** | Creates a copy of the given pen data memory block |
| **EndEnumStrokes** | Unlocks the specified memory block |
| **GetPenDataInfo** | Gets the status information for the pen data memory block |
| **GetPenDataStroke** | Gets the raw data for a stroke stored in the pen data memory block |
| **GetPointsFromPenData** | Returns an array of points from the pen data memory block |
| **MetricScalePenData** | Converts pen data points to one of the supported metric modes |
| **OffsetPenData** | Offsets pen data points by a specified amount |
| **ResizePenData** | Stretches or shrinks the physical size of the pen data memory block |

## Custom Recognizer Functions

The recognizer component is replaceable. You can substitute any vector-based recognizer for the one supplied by Microsoft. *Vector* refers to the recognition of a sequence of pen-location data points collected as the pen is moving. The other method of recognition, *bitmap,* refers to optical character recognition (OCR) techniques, which accept an image of the characters to be recognized as input.

Bitmap recognition is not supported by the Pen API; however, it is possible to write an OCR application under Windows. For a complete discussion of recognition features, see Chapter 7, "Replaceable Components: Recognizers and Dictionaries."

The following table describes the functions that must be supplied in an OEM recognizer DLL.

| Function | Description |
|---|---|
| **CloseRecognizer** | Called when the recognizer is uninstalled by an application |
| **ConfigRecognizer** | Sets recognizer-specific parameters |
| **InitRecognizer** | Gives the recognizer a chance to do any initialization before receiving the first request for recognition |
| **RecognizeDataInternal** | Begins sampling pen data stored in the buffer and converts the input to recognized symbols |
| **RecognizeInternal** | Begins sampling pen data from the tablet and converts the input to recognized symbols |
| **TrainContextInternal** | Informs the recognizer at the DLL recognition level that the raw data input represents the symbol value results using contextual information |
| **TrainInkInternal** | Informs the recognizer at the DLL recognition level that the raw data input represents the symbol value results |

Any recognizer will also make extensive use of the Pen Module functions described in the next section. The Pen Module functions provide the means of capturing and manipulating pen data in the recognition process.

A recognizer may also allow another module to examine the results of recognition before the results are sent to the application. For more details, see the entry for the **SetRecogHook** function in Chapter 9, "Pen API Reference."

## Pen Module Functions

The pen module is the component of the RC Manager that the pen driver calls to enter new events. The pen module is responsible for buffering events and generating the appropriate Windows mouse events.

A pen module operates in two modes, pen collection mode and mouse mode. The module is initialized in mouse mode. The RC Manager handles the switching of the two modes.

The pen module functions fall into two sets. One set consists of the data collection and pen information functions called by recognizers and applications. The other set consists of the functions called by the pen driver.

| Function | Description |
|---|---|
| **AddPenEvent** | Adds a pen event to the pen module for later processing. Called by a pen driver. |
| **EndPenCollection** | Forces an end to pen data collection mode. Called by a recognizer. |
| **GetPenHwData** | Gets data from the internal pen buffer. Called by a recognizer. |
| **GetPenHwEventData** | Gets pen data associated with a range of specified mouse events. Called by an application. |
| **IsPenEvent** | Checks to see if the given mouse event was generated by the pen driver. Called by an application. |
| **ProcessPenEvent** | Tells the pen module to process any pending pen events. Called by a pen driver. |
| **UpdatePenInfo** | Updates the **PENINFO** data structure. Called by the pen driver only when a **PENINFO** value changes. |

## Pen Driver Functions

A pen driver provides the same functionality as the mouse driver in the system, and it provides additional support needed for recognition. A pen driver is free to do any low-level filtering necessary to correct for hardware problems. This includes correcting for pen skip (pen up and down actions due to contact bounce), removing duplicate points, and smoothing jitter. The pen driver is loaded as an installable device driver under Windows version 3.1.

The pen driver provides two types of functionality:

- The hardware-specific code necessary to convert the tablet's data into the standard pen format
- The APIs to examine the capabilities of the tablet and modify some of its behavior

You can find the source code for the pen driver for the Microsoft Mouse and the Wacom tablet in the \PENSDK\DDK\TABLET subdirectories.

The following minimum assumptions apply to the tablet and pen supported by the system:

- The tablet should have a sampling resolution of at least 200 dots per inch. A minimal sampling rate of 120 samples per second is required.
- A barrel button is recommended. The barrel button does not need to be located physically on the pen, but the handling of any barrel events should be in the pen driver.
- The ability to detect the position of the pen when it is not in contact with the tablet surface is strongly recommended.

A pen device can provide capabilities in addition to these minimum requirements. Besides capturing the (x,y) data, a pen device can support as many as eight more types of input data: pressure (force), height, angle, and others. This optional information is passed in with the (x,y) data with each pen packet.

All pen driver functionality is implemented with installable driver messages. A pen driver must also support the standard installable device driver messages.

Appendix B, "Pen Notes for the Windows 3.1 Software Development Kit API," describes the messages and installable device driver interface of the pen extensions.

## Display Driver Functions

The definition of a Windows display driver has been expanded to support inking. Inking is drawing done by the display driver at interrupt time. The mechanisms for managing inking are very similar to those used for managing a cursor, the other Windows mechanism for drawing on the screen at interrupt time without calling the Windows Graphic Device Interface.

The **InkReady** function tells the display driver that ink is ready to be drawn. This API is never called directly by an application. See the Windows Device Driver Development Kit documentation for more details about building a display driver.

A pen display driver must also define a new pen cursor type, IDC_PEN (value 32631).

## Dictionary Functions

When the recognizer has finished attempting to identify the input, it returns a symbol graph to the RC Manager. If the application has requested the NULL dictionary, it generates a symbol string from the symbol graph. Otherwise, the RC Manager invokes a specified set of dictionary modules to perform further processing.

Microsoft provides a dictionary of language words specific to the version of Windows (English, French, and so on), but you can also build your own custom dictionary with the **DictionaryProc** function. If you plan to build your own dictionaries, see Chapter 10, "Pen Structures." Dictionary return codes are described in Chapter 11, "Pen Messages and Constants." For complete details on using the **DictionaryProc** function, see Chapter 9, "Pen API Reference." The **DictionarySearch** function performs the requested dictionary search.

A sample dictionary DLL, \PENSDK\SAMPLES\EXPENSE, is included with this SDK. It is discussed in detail in Chapter 7, "Replaceable Components: Recognizers and Dictionaries."

# Pen API Reference

This chapter describes the Pen Application Program Interface (API) functions, listed alphabetically. Each entry includes a complete description of the function, along with its syntax.

Each API function description contains the following information.

| Heading | Meaning |
|---|---|
| Action | A short description of the function |
| Syntax | The syntax for the API function with a list of parameters |
| Module | The module that contains or provides the function |
| Called By | A list of modules that call this function |
| Comments | Additional information about the function |
| Return Value | The value returned by the function |
| See Also | Related API functions |

The Called By and Module headings tell you what class of software developers will use a particular API function. For example, if you are writing a recognizer, you use functions called by the recognizer or in the recognizer module. If you are writing an application, you use the functions called by an application.

Chapter 10, "Pen Structures," describes the structures used in pen computing.

Chapter 11, "Pen Messages and Constants," describes the messages and constants used in pen computing.

# AddPenEvent

**Action** Adds a pen event to the pen module for later processing

**Module** PenModule

**Called By** Pen driver

**Syntax**

**mov si, offset PenPacket**

**call dword ptr lpfnAddPenEvent**

**Comments** The basic unit of communication between the pen driver and Windows is a *pen packet.* 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. Many physical events—interrupts—may be needed to construct a single logical event.

On entry, DS:SI should point to a pen packet.

The **PENPACKET** structure is defined in the TABLET.INC file. The definition is shown in the following code fragment.

```
; PENPACKET - what drivers should use to communicate with
; PenWin.Dll
;

PENPACKET struc
  wTabletX    dw  0        ; X in tablet coordinates
  wTabletY    dw  0        ; Y in tablet coordinates
  wPDK        dw  0        ; various status bits for packet
  rgwOemData dwMAXOEMDATAWORDS dup (0)  ; OEM info like pressure
PENPACKET ends
```

This function adds the pen packet at ds:[si] to a safe internal location in the pen module. **AddPenEvent** is not reentrant. Once the pen packet has been added to the pen module, the **ProcessPenEvent** function will process any pending pen events. **ProcessPenEvent** can be reentered.

A pen driver should construct a pen packet with tablet interrupts disabled, call **AddPenEvent**, re-enable tablet interrupts, and then call **ProcessPenEvent**.

**Return Value** None

**See Also** **ProcessPenEvent**

# AddPointsPenData

**Action** Adds a set of data points to the pen data structure

**Module** RC Manager

**Called By** Recognizer, application

**Syntax** **HPENDATA AddPointsPenData**(*hpendata, lpPnt, lpvOemData, lpsiNew*)

| Parameter | Type | Description |
|---|---|---|
| *hpendata* | **HPENDATA** | Handle to pen data structure. |
| *lpPnt* | **LPPOINT** | New data points to be added to structure. Zero points may be added to force a change of pen state or to set a new pen state. |
| *lpvOemData* | **LPVOID** | OEM data; can be set to NULL if there is no additional OEM data. This is interpreted as indicated in the pen data header. |
| *lpsiNew* | **LPSTROKEINFO** | Header for new stroke data. Contains the count of points from *lpPnt* to be added. |

**Comments** In use, a call to **GetPenHwEventData** gets the *lpsiNew* and *lpvOemData* values. A subsequent call to **AddPointsPenData** adds these points to the pen data structure. This SDK includes an example program, \PENSDK\SAMPLES\SREC\SREC.C, that illustrates the use of **AddPointsPenData**.

**AddPointsPenData** does not scale the data points. It is the caller's responsibility to ensure the proper scaling.

If adding the points to be added represents the same pen up or down state, the data is appended to the last stroke. Otherwise, a new stroke is added.

**Return Value** **AddPointsPenData** returns a handle to the pen data structure. Normally, this is the same handle that was originally passed to the function. NULL is returned on an error. The size of *hpendata* is limited to 64K.

**See Also** **CreatePenData, GetPenHwData, GetPenHwEventData**

# AtomicVirtualEvent

**Action** Locks out pen packets

**Module** RC Manager

**Called By** System applications (Pen Palette)

**Syntax** **void AtomicVirtualEvent**(*fBegin*)

| Parameter | Type | Description |
|---|---|---|
| *fBegin* | **BOOL** | Set TRUE to begin lockout, Set FALSE to end lockout |

**Comments** This function 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. For 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 );
```

Calling **AtomicVirtualEvent** with a TRUE value blocks out input from physical devices until they are freed with a FALSE call. Therefore, a call to **AtomicVirtualEvent**(FALSE) should quickly follow.

**Return Value** None

**See Also** **PostVirtualKeyEvent, PostVirtualMouseEvent**

# BeginEnumStrokes

**Action** Locks an **HPENDATA** data structure in memory in preparation to enumerating strokes

**Module** RC Manager

**Called By** Application

**Syntax** **LPPENDATA BeginEnumStrokes**(*hpendata)*

| Parameter | Type | Description |
|---|---|---|
| *hpendata* | **HPENDATA** | Handle to pen data structure |

**Comments** This function should be called before **GetPenDataStroke**. The return value from **BeginEnumStrokes** can be used in the **GetPenDataStroke** function. Any **HPENDATA** locked using **BeginEnumStrokes** must be unlocked using **EndEnumStrokes**.

**Return Value** A far pointer of the type **LPPENDATA**. This function will return NULL if *hpendata* is compressed or if the handle cannot be locked.

**See Also** **EndEnumStrokes, GetPenDataStroke**

# BoundingRectFromPoints

**Action** Returns the bounding rectangle specified by the given points

**Module** RC Manager

**Called By** Application, recognizer

**Syntax** **void BoundingRectFromPoints**(*lpPnt*, *cPnt*, *lprectBound*)

| Parameter | Type | Description |
|---|---|---|
| *lpPnt* | **LPPOINT** | Pointer to an array of points |
| *cPnt* | **int** | Count of points |
| *lprectBound* | **LPRECT** | Pointer to bounding rectangle containing *lpPnt* points |

**Comments** Parameter validation does nothing if *lpPnt* or *lprectBound* is an invalid pointer value.

**Return Value** Returns bounding rectangle in *lprectBound*. The rectangle is set to an empty rectangle if *cPnt* is zero.

**See Also** **DPtoTP**

# CharacterToSymbol

**Action** Converts an ANSI string to an array of symbol values

**Module** RC Manager

**Called By** Application

**Syntax** **int CharacterToSymbol**(*lpstr*, *cSyv*, *lpsyv*)

| Parameter | Type | Description |
|---|---|---|
| *lpstr* | **LPSTR** | Pointer to a null-terminated ANSI string |
| *cSyv* | **int** | Count of maximum symbols *lpsyv* can hold |
| *lpsyv* | **LPSYV** | Pointer to symbol values buffer |

**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.

**Return Value** Number of characters converted

**See Also** **SymbolToCharacter**

# CloseRecognizer

**Action** Informs the recognizer that it is being unloaded by an application

**Module** Recognizer

**Called By** RC Manager

**Syntax** **void CloseRecognizer()**

**Comments** This function is called when an application uninstalls a recognizer by a call to **UninstallRecognizer**.

Because multiple applications can load a given recognizer, it is the recognizer's responsibility to maintain a use count to determine whether a given **CloseRecognizer** call is being made because the last application is unloading.

Because an application may abnormally end without unloading a recognizer, the use count may be invalid. An alternate strategy is to perform all necessary closing operations—for example, saving newly trained prototypes—on each call to this function.

This function is in the recognizer and is not called directly by an application.

**Return Value** None

**See Also** **InitRecognizer, InstallRecognizer, UninstallRecognizer**

# CompactPenData

**Action** Compacts the pen data based on specified trim options

**Module** RC Manager

**Called By** Application

**Syntax** **HPENDATA CompactPenData**(*hpendata, wTrimOptions*)

| Parameter | Type | Description |
|---|---|---|
| *hpendata* | **HPENDATA** | Pen data |
| *wTrimOptions* | **UINT** | Data-trimming options |

**Comments** **CompactPenData** trims the pen data in *hpendata* based on the *wTrimOptions* parameter. For complete details on compression and inking, see Chapter 4, "Managing Ink in Windows for Pens Applications."

The following table lists the trim options.

| Trim options | Meaning |
|---|---|
| PDTT_DEFAULT | Reallocates memory block to fit the data; does not trim the data. If you call **CompactPenData** with this trim option and the call **GlobalSize**(hpendata), you can get the size of the data file to be saved. This data can be written out like any other block of binary data. |
| PDTT_ALL | Removes the **PENINFO** structure from the header; throws out all data from up points (points collected when the pen is not in contact with the tablet) and removes OEM data and collinear points. |
| PDTT_COLINEAR | Removes successive identical points and colinear points from the pen data. After the operation is performed, PDTS_NOCOLINEAR is set in the *wPndts* field of the pendata header. The colinear points can be removed with very little loss of recognition accuracy. If the colinear points are removed before the points are scaled to display coordinates, there may be some slight change in visual image. |
| PDTT_COMPRESS | Compresses the data without losing any information. Once the data has been compressed, the compressed *hpendata* can be passed as a parameter only to the following functions: **CompactPenData**, **GetPenDataInfo**, **DuplicatePenData**.<br><br>This is the best compression currently provided by the **CompactPenData** function that retains the ability for an application to come back later and recognize the ink. |

| Trim options | Meaning |
|---|---|
| PDTT_DECOMPRESS | Decompresses the data. You cannot use this option with any other trim options. Compression will restore the data completely. |
| PDTT_OEMDATA | Removes OEM data. (That is, data other than x,y coordinates — for example, pressure.) This option will not affect delayed recognition unless you are using a recognizer that expressly requires OEM data. For example, signature recognizers often use pressure information. |
| PDTT_PENINFO | Removes the **PENINFO** structure from the header. Use this option if there is no OEM data associated with the data points or if the application will not be using any of the OEM data. This option will have no effect on the pen data for delayed recognition. |
| PDTT_UPPOINTS | Throws out all data from up points (points collected when the pen is not in contact with the tablet). This option will have no effect on delayed recognition. It should not be necessary, because the majority of the time up points are not a part of standard pen data. |

**Return Value**

If successful, this function returns an **HPENDATA** handle to the **PENDATA** structure. Otherwise, it returns NULL. **CompactPenData** may fail and return NULL in low memory situations if compression or decompression is requested.

The PDTS_ bits are set in the *wPdts* field in the **PENDATA** header to indicate which operations have been performed.

**See Also**

**CreatePenData**

# ConfigRecognizer

**Action** Sets recognizer-specific parameters

**Module** Recognizer

**Called By** System applications (Pen Palette, Control Panel, and others)

**Syntax** **UINT ConfigRecognizer**(*wConfigRecog*, *wParam*, *lParam*)

| Parameter | Type | Description |
|---|---|---|
| *wConfigRecog* | **UINT** | Specifies recognizer subfunction. |
| *wParam* | **UINT** | See descriptions following. |
| *lParam* | **LONG** | Far pointer to buffer. |

**Comments** This function provides various initializations and query functions. The value of *wConfigRecog* determines what function is performed.

| Recognizer subfunction | Meaning |
|---|---|
| WCR_CONFIGDIALOG | This subfunction informs 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 requests a printer setup.) Some examples of the type of settings a recognizer may choose to 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.<br><br>The *lParam* parameter points to the name of the currently selected user in the Control Panel. Because the global RC may not yet have been updated, this may not be the same as the user in the global **RC**. The *wParam* parameter is used by the recognizer as the parent window for any dialog boxes it displays. The return value is always TRUE.<br><br>A recognizer may also choose to implement this as a NULL procedure consisting of just a **return** statement. |
| WCR_DEFAULT | This subfunction returns TRUE if the recognizer is capable of being a default recognizer. A default recognizer must support the standard character set as well as the gestures. |
| WCR_PRIVATE | Values above WCR_PRIVATE have a recognizer-dependent meaning. |
| WCR_QUERY | This subfunction returns TRUE if the recognizer supports a configuration dialog. |

| Recognizer subfunction | Meaning |
|---|---|
| WCR_QUERYLANGUAGE | The *wParam* parameter is not used. The *lParam* parameter points to a null-terminated language string. For a description of the language strings, see the documentation for the **lpLanguage** field of the **RC** structure. The return value is TRUE if the recognizer supports the language; otherwise, it is FALSE. |
| WCR_RCCHANGE | The *lParam* parameter points to the new default **RC** structure. An application that loads nondefault recognizers should call this subfunction for each recognizer it has loaded in response to a WM_GLOBALRCCHANGE message. The return value is TRUE. A recognizer may support only some **RC** fields, such as language or user fields, at the global level. |
| WCR_RECOGNAME | This subfunction treats the *lParam* parameter 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 description of the recognizer that the Control Panel presents to the user, for example, "US English character set, cursive & print." The return value is always TRUE. |
| 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. 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_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 calling this subfunction is:<br><br>ConfigRecognizer(WCR_TRAINCUSTOM, *hwnd*, *lprcresult*)<br><br>The *hwnd* parameter is a handle to the requesting window; the trainer uses this as the parent window for a dialog box, for example. If there has been a recent recognition, a pointer to it will be passed in the *lParam* field, although this may be null. |

| Recognizer subfunction | Meaning |
|---|---|
| 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 format for this subfunction call is:<br><br>ConfigRecognizer(WCR_TRAINDIRTY,0,0). |
| WCR_TRAINMAX | The recognizer returns the maximum number of SYVs that it can train for any given shape. It should return 0 if it can train any number of characters. For example, the Microsoft recognizer can train as many as three characters for a shape (so that it can recognize ligatures), but a cursive recognizer may allow more. |
| WCR_TRAINSAVE | The trainer calls **ConfigRecognizer** ( WCR_TRAINSAVE, TRAIN_SAVE, 0 ) when it is time to save the database. 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.<br><br>The trainer calls **ConfigRecognizer** ( 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. Again, the recognizer should return TRUE if it is successful. |
| WCR_USERCHANGED | This subfunction notifies the recognizer that a user has changed. The *lParam* parameter points to the name of the user that is affected. The modification is indicated by the *wParam* parameter. Currently, only one modification notification is sent to a recognizer. A *wParam* value of CRUC_REMOVE indicates that the user is being removed from the user list. If the recognizer has saved any files or settings for the user, they should be deleted in response to this notification. If the recognizer receives this message with a new user, it should do nothing.<br><br>The return value is 0. |
| WCR_VERSION | The function returns version number. The low-order byte of the return value specifies the major (version) number. The high-order byte specifies the minor (revision) number. |

**Return Value** The return value is described in the preceding table for each WCR_ value.

# CorrectWriting

**Action** Sends text to the edit text dialog for modification

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL CorrectWriting**(*hwnd, lpBuf, cbBuf, lprc, dwCwrFlags, dwReserved*)

| Parameter | Type | Description |
|---|---|---|
| *hwnd* | **HWND** | Handle to the window that is the owner of the correction dialog. |
| *lpBuf* | **LPSTR** | Text to be corrected. |
| *cbBuf* | **UINT** | Number of characters (greater than 1) in *lpBuf*. This should include space for the NULL terminator of the string. |
| *lprc* | **LPRC** | Pointer to **RC** structure for context or NULL. |
| *dwCwrFlags* | **DWORD** | Translation and style flags. NULL is the default. |
| *dwReserved* | **DWORD** | Reserved for future use. NULL is the default. |

**Comments** An application calls **CorrectWriting** to send text to the Edit Text dialog box. Normally, this would be in response to receiving the SYV_CORRECT gesture.

The *hwnd* parameter contains a Windows handle to the window that will be used as the owner of the dialog box created for correction. The Edit Text dialog box will be application-modal, based on the *hwnd* parameter.

The *lpBuf* parameter contains the text to be corrected. It is zero-terminated with a maximum size of *cbBuf*. The corrected text is also returned in *lpBuf*. As a general rule, the *lpBuf* parameter should allow for growth by a factor of two or more—or some maximum size that is dependent on the field of entry.

If *lprc* is not NULL, the following fields from *lprc* are used as the context for correction. If the field in *lprc* has a default value, it will not override the corrector's default value.

- **lpfnYield**
- **lpLanguage**
- **wTryDictionary**
- **alc**
- **rgbfAlc**
- **dwRecognizer**
- **lpUser**
- **rglpdf**
- **clErrorLevel**
- **alcPriority**
- **dwDictParam**
- **hrec**

The *dwCwrFlags* parameter can be any combination of the following. The default value of NULL indicates that none of these translation flags are to be applied.

| Subfunction | Meaning |
|---|---|
| CWR_STRIPLF | Removes the LF characters |
| CWR_STRIPCR | Removes the CR characters |
| CWR_STRIPTAB | Removes the tab characters |
| CWR_SINGLELINEEDIT | CWR_STRIPLF I CS_STRIPCR I CWR_STRIPTAB |
| CWR_TITLE | If this flag is set, *dwReserved* is interpreted as an LPSTR pointer to a null-terminated caption string. |

If the user chooses OK from the Correction dialog box, and if any of the *dwCwrFlags* (except CWR_TITLE) are set, then the *dwCwrFlags* characters are removed from the buffer before the text is returned to the application.

**Return Value** This function returns TRUE if the OK button is pressed in the Edit Text dialog box. Otherwise, it returns FALSE. If FALSE is returned, the application should leave the text unchanged in the original application.

# CreatePenData

**Action** Creates a **PENDATA** structure

**Module** RC Manager

**Called By** Application, recognizer

**Syntax** **HPENDATA CreatePenData**(*lppeninfo, cbOemData, wPdtScale, gmemFlags*)

| Parameter | Type | Description |
|---|---|---|
| *lppeninfo* | **LPPENINFO** | Far pointer to tablet information for filling the **HPENDATA** structure. If this is NULL, the current tablet settings are used. |
| *cbOemData* | **int** | Width of OEM data packet. If this value is greater than or equal to 0, the OEM data will override whatever is contained in the **PENINFO** structure. |
| *wPdtScale* | **UINT** | Data scaling metric value. (See the following table.) |
| *gmemFlags* | **UINT** | Parameter for specifying the flags passed to the Windows **GlobalAlloc** function when memory for the **HPENDATA** object is created. The *gmemFlags* parameter should be 0 or GMEM_DDESHARE. |

**Comments** The following table lists the *wPdtScale* values. These scaling values are also used in the API function **MetricScalePenData** and in the *wPdts* field of **PENDATAHEADER**.

The scaling values do not behave in the same way as the Windows terms of the same name. For example, in the Windows GDI, a line 1 inch long in MM_HIENGLISH will not necessarily be an inch long on the screen, because the GDI does not know the size of the monitor. However, with PDTS_HIENGLISH in **MetricScalePenData**, an application can assume that a line drawn an inch long really is an inch long.

| Scaling value | Meaning |
|---|---|
| PDTS_LOMETRIC | Each logical unit is mapped to 0.01 mm. Positive x is to the right; positive y is down. |
| PDTS_HIMETRIC | Each logical unit is mapped to 0.001 mm. 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_TABLET | Data points are in tablet units. |

| Scaling value | Meaning |
|---|---|
| PDTS_ARBITRARY | The application has done its own scaling of the data point. |
| PDTS_STANDARDSCALE | The standard scaling metric is equivalent to PDTS_HIGHENGLISH. |

**Return Value** This function returns a handle to the pen data memory block that was created. The **HPENDATA** data type is returned to the recognizer to represent raw data. On creation, the pen data memory block contains no point.

If successful, this function returns an **HPENDATA** handle to the **PENDATA** structure. Otherwise, it returns NULL. It returns NULL on a memory allocation failure.

**See Also** **DestroyPenData**

# DestroyPenData

**Action** Frees up memory associated with an *hpendata* memory block

**Module** RC Manager

**Called By** Application, recognizer

**Syntax** **#define DestroyPenData**(*hpendata*) (**GlobalFree** (*hpendata*) == NULL)

**Comments** The **DestroyPenData** macro uses the Windows **GlobalFree** function to free up memory associated with the specified *hpendata* memory block.

**Return Value** This function returns TRUE if the macro is successful. Once memory is freed, the *hpendata* handle is no longer valid.

**See Also** **CreatePenData**

# DictionaryProc

**Action** Entry point into dictionary module

**Module** Dictionary

**Called By** RC Manager

**Syntax** **int FAR PASCAL DictionaryProc**(*dirq, lpIn, lpOut, cbMax, dwContext, dwData*)

| Parameter | Type | Description |
|---|---|---|
| *dirq* | **int** | Dictionary subfunction number. See the dictionary subfunction table following. |
| *lpIn* | **LPVOID** | Input parameter (type dependent on *dirq*). |
| *lpOut* | **LPVOID** | Output parameter (type dependent on *dirq*). |
| *cbMax* | **int** | Size of *lpOut* buffer. |
| *dwContext* | **DWORD** | Reserved for future implementation. |
| *dwData* | **DWORD** | Dictionary-specific data. |

**Comments** This is the only exported function required in any dictionary DLL. A sample dictionary DLL is discussed in Chapter 7, "Replaceable Components: Recognizers and Dictionaries." A sample dictionary is supplied with the SDK. It is available in the \SAMPLES\EXPENSE subdirectory.

The process of loading dictionaries and identifying entry points is handled in the same way as any other DLL. That is, you use the **LoadLibrary** and **GetProcAddress API** functions under Windows.

## Dictionary Subfunction Table (dirq Parameter)

The following table lists the dictionary subfunctions.

| DIRQ subfunction | Meaning |
|---|---|
| DIRQ_QUERY | This subfunction queries the dictionary. The *lpIn* parameter is a far pointer to an integer containing a DIRQ_ value. It returns TRUE if the dictionary supports the subfunction; otherwise, it returns FALSE. |

| DIRQ subfunction | Meaning |
| --- | --- |
| DIRQ_STRING | This subfunction looks for the existence of a string. The *lpIn* parameter points to a SYV_NULL-terminated array of symbol values. If a match for the string is found, *cbmax* symbols from *lpIn* are returned in *lpOut*. If not successful, DIRQ_STRING returns zero. Otherwise, it returns the number of symbols copied to *lpOut*. SYV_NULL terminates the output buffer if enough space is available. The return count does not include SYV_NULL.<br><br>The *lpIn* parameter can be NULL. NULL can be used as a signal to the dictionary to return any deferred match.<br><br>Note that a dictionary implementation does not have to be a simple lookup. For example, a dictionary function can be written to determine if the input matches the format for a social security number. |
| DIRQ_SUGGEST | This subfunction is provided so that applications can use the same dictionaries for spell checking as for handwriting recognition. The *lpIn* parameter is a pointer to a symbol string. Symbol strings terminated by SYV_NULL are placed in *lpOut*. The end is marked by two successive SYV_NULLs. The return value is the count of alternatives placed in *lpOut*. A dictionary implementation can be written so that, if the word being looked up is in the dictionary, DIRQ_SUGGEST always returns FALSE or, alternatively, the contents of *lpIn* in *lpOut*. |
| DIRQ_SYMBOLGRAPH | This subfunction is provided so that dictionaries can examine the whole symbol graph before each enumeration is passed in individually. The *lpIn* parameter contains a far pointer to the symbol graph. The other parameters are the same as those in DIRQ_STRING. Before enumerating, DIRQ_SYMBOLGRAPH checks each dictionary once to see if any dictionary is prepared to handle the whole symbol graph. The check takes place if RCO_SUGGEST is on. |
| DIRQ_USER | Values greater than DIRQ_USER (that is, >4096) can be used for private calls to the dictionary DLL. Each dictionary can define its own values. |

Other DIRQ_ values are defined for adding words, deleting words, and saving dictionaries. These are discussed in the OEM Dictionary Module section, later in this listing. However,

only the subfunctions in the preceding list are called by the **DictionarySearch** API, which is in the RC Manager.

A dictionary called with a subfunction number it does not support should return FALSE. A dictionary must support DIRQ_OPEN, DIRQ_CLOSE, DIRQ_QUERY, DIRQ_STRING, DIRQ_DESCRIPTION, DIRQ_INIT, and DIRQ_CLEANUP.

## NULL Dictionary

The NULL dictionary provides the minimum mapping functionality from a symbol graph to a symbol string. The resulting string is the best enumeration as returned by **FirstSymbolFromGraph**.

## DictionarySearch Function

When the recognizer has attempted to identify the input, it returns a symbol graph. If the application has requested the NULL dictionary, it generates a symbol string from the symbol graph. Otherwise, the RC Manager calls the **DictionarySearch** function.

If the **RCRESULT** field **hSyv** is not NULL, the RC Manager assumes that the dictionary has already filled it in. This means the recognizer can perform the dictionary processing itself. Once the recognizer has completed its dictionary processing, it simply sets **hSyv** in **RCRESULT** to a non-NULL value. The recognizer must set **hSyv** to NULL every time if it is not performing the dictionary processing.

## Word Definitions and Validation

If you build a dictionary, you determine the rules for identifying which words to validate. For example, the symbol graph may contain punctuation or other symbol values not in the character set supported by the dictionary. A dictionary can strip out punctuation and perform the lookup, or it can just fail on words containing symbols outside the dictionary's supported character set.

Unless RCO_NOSPACEBREAK is set in the **RC** structure, the symbol strings passed to the dictionaries are divided at white-space characters.

A symbol graph passed to a dictionary may include a space as an option. For example, the symbol graph `fat{space|null}cat` represents two possible strings: "fat cat" and "fatcat." These are passed in as two words.

The RC Manager passes a parameter into the dictionary to provide access to dictionary functionality not directly supported. The *dwDictParam* parameter is set by an application and passed on to the dictionary by the RC Manager. For example, a dictionary may request that the application pass in a pointer to a structure containing the sentence to which the results of the recognition will be added, as well as an indication of where in the sentence the new characters are to be added.

## Spell Checking

Spell checking should be handled in the normal ways by the application using handwriting input. Dictionary modules can also be used by the recognizer for spell checking. However, to avoid outguessing the user, the recognizer does not attempt corrections based on

spelling errors except when DIRQ_SUGGEST is specified. For example, if the user writes "seperate," the recognizer will not report "separate"—unless it indicates that the second "e" could be an "a." However, the application is free to take the result returned by the RC Manager and perform spell checking on it.

## OEM Dictionary Module

The dictionary module provides a way of loading in DLLs as dictionaries and establishing a default dictionary to look through. Loading dictionaries and identifying entry points are handled in the same way that printer drivers use the Windows **LoadLibrary** and **GetProcAddress** functions.

| DIRQ subfunction | Meaning |
|---|---|
| DIRQ_ ADD | Adds a word to the word list. The *lpOut* parameter is the word (SYV string) to add. The *lpIn* parameter is a handle to the word list returned by DIRQ_OPEN.<br><br>DIRQ_ADD returns 1 if the word if successfully added. Otherwise, it returns 0. |
| DIRQ_CLOSE | Closes or discards a word list. The *lpIn* parameter is a pointer to the handle previously received after a successful DIRQ_OPEN.<br><br>DIRQ_CLOSE returns 1 if successful; otherwise, it returns 0.<br><br>The dictionary DLL determines whether or not to save all of the changes made through DIRQ_ADD, DIRQ_DELETE, and DIRQ_FLUSH before closing a word list. |
| DIRQ_CONFIGURE | Opens the dictionary DLL's private configuration dialog. This can be used to establish which word lists to use with the given dictionary DLL. The *lpIn* parameter points to a window handle to be used as the parent of the dialog box.<br><br>DIRQ_CONFIGURE returns 1 if the user makes a change. |
| DIRQ_COPYRIGHT | Returns in *lpOut* a maximum of *cbMax* characters of a copyright notice for the dictionary DLL. This subfunction returns TRUE if if *lpOut* is not empty. The description should not exceed 255 bytes. |
| DIRQ_DELETE | Deletes a word from the word list. The *lpOut* parameter is the word (SYV array) to delete. The *lpIn* parameter is a word list handle.<br><br>DIRQ_DELETE returns 1 if the word is successfully deleted. |

| DIRQ subfunction | Meaning |
|---|---|
| DIRQ_DESCRIPTION | Returns in *lpOut* a description string for the dictionary DLL. Example: "Microsoft English Language Dictionary." This description is used by the Control Panel. The description should not exceed 80 bytes. |
| DIRQ_FLUSH | Deletes all words in a word list. The *lpIn* parameter points to the handle of the word list. |
| DIRQ_OPEN | Loads a word list. The *lpIn* parameter points to a string specifying the word list to load. It can be a full path or any other implementation-dependent method of identifying individual word lists.<br><br>Dictionary word-list files are dependent on the particular dictionary implementation. Dictionary implementation determines whether successive loads are merged or replace one another.<br><br>If the subfunction is successful, the *lpOut* parameter points to the handle to the word list for subsequent use.<br><br>DIRQ_OPEN returns TRUE if successful. |
| DIRQ_QUERY | Queries the dictionary. The *lpIn* parameter points to an integer representing a DIRQ_ value.<br><br>DIRQ_QUERY returns TRUE if the dictionary supports the subfunction. |
| DIRQ_RCCHANGE | Enables dictionaries to respond to a change in their environment—for example, a .INI change of current language. This subfunction is also used to allow a change in any other parameter in the supplied RC. It is the responsibility of an application to call this subfunction for all nondefault dictionaries it loads. It is analogous to the WCR_RCCHANGE subfunction of **ConfigRecognizer**.<br><br>The *lpIn* parameter points to the new global RC.<br><br>Language dictionary DLLs can unload the currently loaded language dictionary and load the new language dictionary in response to this call. How or whether to respond to this call is determined by the DLL implementation. |

| DIRQ subfunction | Meaning |
|---|---|
| DIRQ_SETWORDLISTS | Sets the user/language word lists to search through in DIRQ_STRING, DIRQ_SUGGEST, and DIRQ_SYMBOLGRAPH commands. If the function is not successful because of incorrect handles, the old search lists are retained.<br>The *lpIn* parameter contains an array of handles obtained by using DIRQ_OPEN. The *cMax* parameter is the count of handles in the array.<br>DIRQ_SETWORDLISTS returns TRUE if successful. |

## Microsoft User Dictionary DLL

The Microsoft User Dictionary DLL (USERDICT.DLL) supports a user-defined set of words, with the following DIRQ_ values.

- DIRQ_ADD
- DIRQ_CLOSE
- DIRQ_DELETE
- DIRQ_DESCRIPTION
- DIRQ_OPEN
- DIRQ_QUERY
- DIRQ_SETWORDLISTS
- DIRQ_STRING

On initialization (in **LibMain**), the user dictionary DLL loads all of the user dictionaries (to a maximum of 16) in the [MsUserDict] section of the PENWIN.INI file.

The PENWIN.INI file might look like the following:

```
[MsUserDict]
c:\pensdk\bin\names.dic=
c:\states.dic=
```

Each task can open a maximum of 16 word lists, so as many as 16 word lists can be set up for default searching. If a task performs a DIRQ_SEARCH without first using a DIRQ_OPEN or a DIRQ_SETWORDLIST, the input buffer will be searched for a match in the default dictionaries.

The word lists are a list of strings, one string per line, listed in alphabetic order. A user dictionary can be created using a word processor. The word-list file should not have any other formatting information and should not be terminated by a ^Z character.

## Microsoft Language Dictionary DLL

The Microsoft Language Dictionary DLL (MAINDICT.DLL) is a multilingual dictionary that currently allows more than one dictionary to be loaded at a time.

MAINDICT.DLL supports the following DIRQ_ subfunctions:

- DIRQ_CLEANUP
- DIRQ_CLOSE
- DIRQ_COPYRIGHT
- DIRQ_DESCRIPTION
- DIRQ_INIT
- DIRQ_OPEN
- DIRQ_QUERY
- DIRQ_RCCHANGE
- DIRQ_SETWORDLISTS
- DIRQ_STRING
- DIRQ_SUGGEST

On initialization (with **LibMain**), MAINDICT.DLL loads the current language dictionaries based on the *sLanguage* keyword in PENWIN.INI.

The following table lists the implementation for each DIRQ_ subfunction within MAINDICT.DLL. Only those subfunctions with additional details beyond the descriptions given previously are described.

| DIRQ subfunction | Description |
|---|---|
| DIRQ_CLEANUP | This subfunction is provided so that dictionaries can do any cleaning up before termination. The RC manager calls dictionary DLLs with this message before unloading the dictionary DLL. |
| DIRQ_CLOSE | Closes a language word list. The *lpIn* parameter is a pointer to the handle received after a successful DIRQ_OPEN.<br>DIRQ_CLOSE returns TRUE if the close is successful. |
| DIRQ_COPYRIGHT | This subfunction returns the copyright string, if one is available, into the buffer specified by *lpOut*. The size of the buffer is *cbMax*. It returns TRUE if a copyright string was returned; otherwise it returns FALSE. Dictionaries should be able to respond to this message at all times. |
| DIRQ_INIT | This subfunction is provided so that dictionaries can do any initialization (such as loading a default word list). The RC manager calls the dictionary DLLs with this subfunction before using them for spell checking during the recognition process.<br>A dictionary DLL may get two or more consecutive DIRQ_INIT messages before getting any DIRQ_CLEANUP messages. You should keep a count of the DIRQ_INIT calls, which are incremented and decremented for each DIRQ_INIT or DIRQ_CLEANUP subfunction use. |

| | |
|---|---|
| DIRQ_OPEN | Opens a language word list. The *lpIn* parameter is a pointer to a three-letter language code for the language word list to be loaded. These codes are listed later in this section. The *lpOut* parameter returns the handle to the language word list. Other parameters are not used.<br><br>The function returns TRUE if the open is sucessful; otherwise, it returns FALSE. The handle returned on a successful open can later be used for setting the word lists to search through. For more information, see DIRQ_SETWORDLISTS, later in this listing.<br><br>Version 1.0 of MAINDICT.DLL can deal with only one language at a time. |
| DIRQ_RCCHANGE | Responds by loading a new language dictionary if the current language has changed and the dictionary is not already loaded. |
| DIRQ_SETWORDLISTS | Sets the language word lists to search through in DIRQ_STRING and DIRQ_SUGGEST commands. If the function is not successful because of incorrect handles, the old search lists are retained.<br><br>The *lpIn* parameter is (**int FAR** *) containing an array of handles obtained by doing DIRQ_OPEN. The *cMax* parameter is the count of handles in the array.<br><br>DIRQ_SETWORDLISTS returns TRUE if successful. |
| DIRQ_STRING | If the calling task has not made any open calls to DIRQ_OPEN, or its search list is empty, the DLL searches through the current language dictionary. Otherwise, it goes through the search lists in sequence. In Windows version 3.1, you can set or change the current language, using the Control Panel. |

## Language Codes for DIRQ_OPEN

The following are the three-letter language codes that *lpIn* can point to when you open a language word list. They are the same as those allowed under Windows, version 3.1.

- NLD (Dutch)
- ENG (International English)
- FRC (French Canadian)
- FRA (French)
- DEU (German)
- ISL (Icelandic; not yet supported)
- ESN (Modern Spanish)
- FIN (Finnish)
- ITA (Italian)
- NOR (Norwegian)
- PTG (Portuguese)
- ESP (Spanish)
- SVE (Swedish)
- ENU (American English)
- DAN (Danish)

To load the language word list, the DLL looks for keyword *XXX***Main=** where *XXX* represents any of the three-letter language codes under the [MsMainDict] section in PENWIN.INI . If the keyword is found, the DLL tries to open the file indicated. Otherwise, it looks for MSSP_YY.LEX in the BIN directory—or wherever you installed the SDK. Note that multilingual searches are costly in terms of time and memory.

The "YY" letters stand for the following language-specific two-letter codes:

- NL (Dutch)
- BR (English)
- FC (French Canadian)
- FR (French)
- GE (German)
- IT (Italian)
- NN (Norwegian)
- PB (Portuguese)
- SP (Spanish)
- SW (Swedish)
- AM (American English)

**See Also** **DictionarySearch**

# DictionarySearch

**Action** Performs the requested dictionary search

**Module** RC Manager

**Called By** Application, RC Manager, recognizer

**Syntax** **BOOL DictionarySearch**(*lprc, lpsye, cSye, lpsyv, cSyvMax*)

| Parameter | Type | Description |
|---|---|---|
| *lprc* | **LPRC** | Pointer to the **RC** structure to be used in the search. |
| *lpsye* | **LPSYE** | Pointer to an array of symbol elements that constitute the symbol graph. |
| *cSye* | **int** | Number of SYEs in the array. |
| *lpsyv* | **LPSYV** | Output buffer of SYVs. This parameter contains the return results of the dictionary search. |
| *cSyvMax* | **int** | Size of the output buffer. |

**Comments** The **DictionarySearch** function takes the symbol graph in *lpsye,* performs a dictionary search based on the options set in *lprc,* and returns the result as an array of SYVs 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. A SYV_NULL value is always appended at the end, and therefore *lpsyv* must have enough space for *csyvMax*+1 SYV elements.

**DictionarySearch** first passes in the symbol graph with DIRQ_SYMBOLGRAPH to all the dictionaries in the **rglpdf** array in the *lprc* structure. If none succeeds, the function enumerates the symbol graph in *lpsye* and searches through all of the dictionary functions for a match. The caller can get suggestions by setting the RCO_SUGGEST flag in the **lRcOptions** field in *lprc*. 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. It 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 **lRcOptions** field of *lprc*, the function 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** field in *lprc* to specify the maximum number of enumerations to search through for each symbol graph token.

**Return Value** The **DictionarySearch** function 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.

**See Also** **DictionaryProc**

# DPtoTP

**Action** Converts an array of points in display coordinates to tablet coordinates

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL DPtoTP**(*lpPnt*, *cPnt*)

| Parameter | Type | Description |
|---|---|---|
| *lpPnt* | **LPPOINT** | Pointer to an array of points |
| *cPnt* | **int** | Count of points |

**Comments** Because of possible rounding errors, the **DPtoTP** and **TPtoDP** functions are not guaranteed to be complete inverses of each other. It is the caller's responsibility to avoid overflow by not passing in points beyond the limits of the current physical display.

**Return Value** **DPtoTP** returns TRUE.

**See Also** **TPtoDP**

# DrawPenData

**Action** Displays the pen data in the specified device context

**Module** RC Manager

**Called By** Application

**Syntax** **void DrawPenData**(*hdc, lprect, hpendata*)

| Parameter | Type | Description |
|---|---|---|
| *hdc* | **HDC** | Device context |
| *lprect* | **LPRECT** | Bounding rectangle for ink, in client coordinates |
| *hpendata* | **HPENDATA** | Handle to pen data memory block |

**Comments** **DrawPenData** draws the pen data specified by *hpendata* into the device context specified by *hdc*. The *hdc* value may also specify a Windows metafile handle.

The application using **DrawPenData** is reponsible for either scaling the data points or setting the mapping appropriately if *lprect* is NULL.

If *lprect* is not NULL, the points are scaled into *lprect* as the drawing is done. Internally, a non destructive use of **SetViewportExt – SetViewPortOrg** and **SetWindowOrg – SetWindowExt** renders the pen data in the device context within the bounds of the provided rectangle. You will have to compute the proper pen width (assuming it is other than 1) before calling this function with a non-null *lprect* to account for the scaling that is going to occur.

**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 HDC's origin by previous calls to the Windows functions **SetWindowOrg** or **SetViewportOrg**. If the origin has changed, the rectangle passed to **DrawPenData** must be offset by the appropriate amount.

The **PENDATA** structure has no graphic attributes —such as ink width or color— associated with it.

If the ink is to be drawn with a width of greater than 1 pixel, the currently selected pen must be set to the appropriate width (in client coordinates if a mapping mode is set in the DC) for the proper width to result. For example, if the mapping mode has been set to MM_HIENGLISH, then the pen width needs to be set to a number appropriate for the desired width in HIENGLISH units to preserve the proper scale of the ink. This scaling is an issue only when the ink width is greater than 1.

Because of aliasing effects, **DrawPenData** may not draw the exact same pixels that were originally inked. If an exact reproduction is required, you should use **RedisplayPenData**.

**Return Value** None. If *hpendata* is NULL, **DrawPenData** does nothing.

**See Also** **CreatePenData, DuplicatePenData, RedisplayPenData**

# DuplicatePenData

**Action** Creates a copy of the pen data

**Module** RC Manager

**Called By** Application

**Syntax** **HPENDATA DuplicatePenData**(*hpendata, gmemFlags*)

| Parameter | Type | Description |
|---|---|---|
| *hpendata* | **HPENDATA** | Data to be duplicated |
| *gmemFlags* | **UINT** | Memory-allocation flag |

**Comments** The **DuplicatePenData** function duplicates the data specified by the *hpendata* parameter by creating a second pen data memory block.

The *gmemFlags* parameter specifies the flags to be passed to the Windows **GlobalAlloc** function when memory for the pen data memory block is created. The *gmemFlags* parameter is connected with GMEM_MOVEABLE by an OR operator when the global heap data is allocated.

**Return Value** If successful, this function returns an **HPENDATA** handle to the duplicate **PENDATA** structure. Otherwise, it returns NULL. It will return NULL on a failure to allocate memory.

**See Also** **CreatePenData, DestroyPenData**

# EmulatePen

**Action** Emulates a pen in applications that do not use the standard Windows I-beam cursor in text areas

**Module** RC Manager

**Called By** Application

**Syntax** **void EmulatePen** (*fPen*)

| Parameter | Type | Description |
|---|---|---|
| *fPen* | **BOOL** | Flag to set pen emulation. TRUE activates pen emulation; FALSE turns it off. |

**Comments** **EmulatePen** enables applications that do not use the I-beam to get I-beam functionality over text fields in the application. Call this function with *fPen* set to TRUE whenever the cursor is over a text window in the application. When the cursor leaves that area, call **EmulatePen** with *fPen* set to FALSE. If *fPen* has been set to TRUE and the Pen Palette is currently running, then a tap of the pen will enable the Pen Palette to process the pen input for the active application. If the Pen Palette is not present, calling **EmulatePen** has no effect.

**Note** **EmulatePen** might not be supported in future versions of Microsoft Windows for Pen Computing. This function is intended to aid those developers using Windows software development tools that have not been updated for pen functionality and that do not give the developers access to Windows handles or other Windows APIs. Most applications can use **ProcessWriting** or hedit controls instead. These methods are described in Chapter 6, "Using Pen Controls and the **ProcessWriting** Function."

**Return Value** None

**See Also** **ProcessWriting**

# EndEnumStrokes

**Action** Unlocks the specified memory block

**Module** RC Manager

**Called By** Application

**Syntax** **#define EndEnumStrokes** (*hpendata*) **GlobalUnlock** (*hpendata*)

**Comments** The **EndEnumStrokes** macro uses the Windows **GlobalUnlock** function to unlock the global memory block specified by the *hpendata* parameter. This macro should be called when the application has completed the **GetPenDataStroke** function. The buffer filled by a call to **GetPenDataStroke** will not be valid after a call to **EndEnumStrokes**.

**Return Value** The function returns NULL if the macro is successful. Once this has been called, any pointers returned by **GetPenDataStroke** are invalid.

**See Also** **BeginEnumStrokes**, **GetPenDataStroke**

# EndPenCollection

**Action** Forces an end to pen collection mode

**Module** RC Manager

**Called By** Recognizer

**Syntax** **BOOL EndPenCollection**(*recEnd*)

| Parameter | Type | Description |
|---|---|---|
| *recEnd* | **REC** | Error code parameter |

**Comments** The **EndPenCollection** function is called by a recognizer or the RC Manager. The recognizer calls this function to stop collection. This may occur, for example, if the recognizer runs out of memory.

The error code parameter, *recEnd,* will be returned by **GetPenHwData** on some future call. Since data might still be available when this call is made, the caller should continue to call **GetPenHwData** until the return value indicates that pen mode has ended.

**Return Value** **EndPenCollection** returns TRUE if the call succeeds. The call will fail if the pen is down when the call is made and *recEnd* is not negative. If the call fails, the pen module remains in the collection mode. This function returns FALSE if it is not currently in collection mode.

# EnumSymbols

**Action** Enumerates strings from a symbol graph in order of most probable to least probable

**Module** RC Manager

**Called By** RC Manager, application

**Syntax** **UINT EnumSymbols**(*lpsyg, wMaxStr, lpEnumFunc, lvData*)

| Parameter | Type | Description |
|---|---|---|
| *lpsyg* | **LPSYG** | Pointer to symbol graph |
| *wMaxStr* | **UINT** | Maximum number of strings to enumerate |
| *lpEnumFunc* | **FARPROC** | Pointer to enumeration function |
| *lvData* | **LPVOID** | Application-specific data |

**Comments** The **EnumSymbols** function enumerates all strings of symbols—to a maximum defined by *wMaxStr*—that can be generated from the symbol graph *lpsyg*. The *lpEnumFunc* parameter is a pointer to the enumeration function called with each enumeration.

To generate all the symbols from a symbol graph, set *wMaxStr* equal to **GetSymbolCount**(*lpsyg*).

## EnumFunc

**int FAR PASCAL EnumFunc**(*lpsyv*, *csyv*, *lvData*)

| Parameter | Type | Description |
|---|---|---|
| *lpsyv* | **LPSYV** | Symbol string |
| *csyv* | **int** | Count of symbols in string |
| *lvData* | **FAR VOID *** | Pointer passed in from **EnumSymbols** |

**EnumFunc** is a placeholder for an application-supplied name. This function must be exported in the module definition file. **EnumFunc** should return 0 to stop enumeration.

**Return Value** **EnumSymbols** returns the number of strings enumerated.

**See Also** **FirstSymbolFromGraph, SYG** structure

# ExecuteGesture

**Action** Executes a gesture

**Module** RC Manager

**Called By** System application (Pen Palette)

**Syntax** **BOOL ExecuteGesture**(*hwnd, syv, lprcresult)*

| Parameter | Type | Description |
|---|---|---|
| *hwnd* | **HWND** | Window handle |
| *syv* | **SYV** | Gesture to map |
| *lprcresult* | **LPRCRESULT** | Pointer to the **RCRESULTS** structure |

**Comments** This function attempts to convert the gesture in *syv* to a set of keystrokes that the user has mapped using the Gesture Mapper. The *hwnd* parameter specifies the application that will receive the mapped gesture. If no mapping exists, **ExecuteGesture** tries to get the default mapping of the gesture. If a mapping is found, one of three things happens:

- The gesture is sent to the system queue as a set of keystrokes.
- The mapping is placed in the *lpsyv* element of the RCRESULTS structure.
- The mapping is placed in the *lpsyv* element as another gesture if it maps to some standard key sequences.

In general, **ExecuteGesture** is called once to get the mapping. If necessary, it is called a second time to send the mapped keystrokes to the application.

If *lprcresult* is NULL, the result is always sent as keystrokes. If *lprcresult* is not NULL, the type conversion is indicated by the flags in the *wResultsType* element of *lprcresult*. If it is translated to an ANSI string, the RCRT_GESTURETRANSLATED flag is set. If it is translated to another gesture, the RCRT_GESTURETRANSLATED and RCRT_GESTURE flags are set. If neither of these two translations is true, the translation is to a set of virtual key codes, and the RCRT_GESTURETOKEYS and RCRT_ALREADYPROCESSED flags are set.

To send the results of a mapping, call **ExecuteGesture** with the RCRT_GESTURETOKEYS flag set in the *wResultsType* field in the RCRESULT structure. The function will send the virtual keys that have been placed in *lpsyv* as symbol values to the system as keystrokes. Once the keys have been sent, **ExecuteGesture** resets the flag.

If the result of the mapping is one of the keystrokes listed in the following table, then the keystrokes are replaced by the corresponding symbol gesture.

| Keystrokes | Gesture |
|---|---|
| CTRL + INS | SYV_COPY |
| SHIFT + INS | SYV_PASTE |
| ALT + BACKSPACE | SYV_UNDO |
| SHIFT + DEL | SYV_CUT |

When a gesture is mapped to virtual keys and the keys are inserted into the *lpsyv* element of the *lprcresult* parameter passed in, each element in the *lpsyv* array is of the type SYV with the following format:

- The HIWORD is set to SYVHI_VKEY.
- If the LOWORD has the most significant bit cleared, the LOBYTE contains the virtual keycode as defined in WINDOWS.H. The HIBYTE has CONTROL, ALT, or SHIFT bits set. For example, 0x00060441 means an SYV for ALT+A, 0x00060270 means CTRL+F1, and 0x0006012D means SHIFT+INS.
- If the LOWORD has the most significant bit set, the lower three nibbles contain binary coded decimal numbers indicating an ALT+NUMPAD combination. For example, 0x00068199 means an SYV for an ALT+NUMPAD combination that translates to ANSI code 199.

**Return Value** This function returns TRUE if the given gesture is mapped to text, keystrokes, or another gesture.

# FirstSymbolFromGraph

**Action** Places into *lpsyv* the array of symbols that is the likeliest interpretation of the symbol graph

**Module** RC Manager

**Called By** Application

**Syntax** **void FirstSymbolFromGraph**(*lpsyg*, *lpsyv*, *cSyvMax*, *lpcSyv*)

| Parameter | Type | Description |
|---|---|---|
| *lpsyg* | **LPSYG** | Symbol graph. |
| *lpsyv* | **LPSYV** | Array of symbols. |
| *cSyvMax* | **int** | Buffer size. |
| *lpcSyv* | **LPINT** | 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. |

**Comments** The array of symbols is identical to the first string returned to the **EnumFunc** procedure of **EnumSymbols**.

**Return Value** None

**See Also** **EnumSymbols**

# GetGlobalRC

**Action** Queries the current default settings and fills the **RC** structure with the global values

**Module** RC Manager

**Called By** System applications (Control Panel)

**Syntax** **UINT GetGlobalRC**(*lprc, lpDefRecog, lpDefDict, cbDefDictMax*)

| Parameter | Type | Description |
|---|---|---|
| *lprc* | **LPRC** | Pointer to **RC** structure. |
| *lpDefRecog* | **LPSTR** | Buffer in which the default recognizer module name is returned. This must be 128 bytes long. |
| *lpDefDict* | **LPSTR** | Buffer in which the default dictionary path is returned. This ends with double zero byte values. |
| *cbDefDictMax* | **int** | Size of *lpDefDict* buffer to be filled. |

**Comments** **GetGlobalRC** fills the **RC** structure with global values. Values that have no default settings—for example, the bounding rectangle—are set to zero.

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 fields. Any of the parameters *lprc*, *lpDefRecog*, or *lpDefDict* may be NULL to indicate that the caller is not interested in the value.

This function returns the actual current values for **RC** fields. The **InitRC** function returns the default values; these include placeholder values for some **RC** fields.

**Return Value** The following table lists the possible return values:

| Value | Meaning |
|---|---|
| GGRC_OK | The function has run successfully without any errors. |
| GGRC_PARAMERROR | One or more invalid parameters were detected. The call to **GetGlobalRC** has no effect. |
| GGRC_DICTBUFTOOSMALL | The size of the *lpDefDict* 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. |

**See Also** **InitRC**, **SetGlobalRC**

# GetMessageExtraInfo

**Action** Gets extra information associated with a mouse message

**Module** Windows (USER.EXE)

**Called By** RC Manager, application

**Syntax** **LONG GetMessageExtraInfo()**

**Comments** The Windows 3.1 **GetMessageExtraInfo** function is called in response to a mouse message to retrieve the extra information passed along with the event. The following paragraphs describe pen-specific information.

If the message was generated by a pen driver, the low word of the return value contains the *wEventRef* value and the high word contains the shift states of the barrel buttons (the PDK_ values). The *wEventRef* value can be used as the parameter to **Recognize** or **GetPenHwEventData** to reference the tablet data.

If the message was not generated by a pen driver, the return value of **GetMessageExtraInfo** should be zero—as it is, for example, for a Microsoft Mouse. However, because this value was undefined in previous versions of Windows, it may not be zero. The **IsPenEvent** function can test the value to determine if it is a pen event even if the long value is nonzero. All devices and emulators that guarantee compatibility with stylus use will set the long value to zero. For other messages, the value is undefined.

Because of the manner in which Windows coalesces mouse messages, an application is guaranteed to get only the most recent position of the mouse during a move—and not necessarily all the intervening positions. Although this is more than acceptable for most uses of the mouse, it can cause problems with recognition efforts.

The pen module maintains a circular buffer of the pen events. As each event is received from the pen driver, it is placed in the buffer as well as entered into the system message queue as a mouse message. Along with the mouse message, two other pieces of information are tagged to the message: the state of the pen driver barrel buttons and a reference to the raw pen event (an index into the buffer).

# GetPenAsyncState

**Action** Gets the barrel button state of the pen

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL GetPenAsyncState**(*wPDK*)

| Parameter | Type | Description |
|---|---|---|
| *wPDK* | **UINT** | PDK_ value |

**Comments** The *wPDK* parameter is one of the PDK_ values for the barrel buttons. The following table lists the PDK_ values.

| Constant | Value | Meaning |
|---|---|---|
| PDK_BARREL1 | 0x0002 | Set if barrel button #1 is depressed |
| PDK_BARREL2 | 0x0004 | Set if barrel button #2 is depressed |
| PDK_BARREL3 | 0x0008 | Set if barrel button #3 is depressed |

**Return Value** This function returns TRUE if the given barrel state is down when the call is made.

# GetPenDataInfo

**Action** Returns the header and pen information from the pen data memory block

**Module** RC Manager

**Called By** Application, RC Manager, recognizer

**Syntax** **BOOL GetPenDataInfo**(*hpendata, lppendataheader, lppeninfo, dwReserved*)

| Parameter | Type | Description |
|---|---|---|
| *hpendata* | **HPENDATA** | Raw pen data |
| *lppendataheader* | **LPPENDATAHEADER** | Buffer for the information contained in the pen data type structure header |
| *lppeninfo* | **LPPENINFO** | Pointer to the PENINFO structure (may be NULL) |
| *dwReserved* | **DWORD** | Reserved for future use. |

**Comments** Applications can use this routine to get the header and pen information in the pen data memory block. If *lppeninfo* is non-NULL and the pen data does not contain pen information, the contents of *lppeninfo* are not changed.

The amount of data allocated is contained in *lppendataheader ->cbSizeUsed.*

**Return Value** This function returns TRUE if successful. The returned data is placed in *lppendataheader*. **GetPenDataInfo** returns FALSE if invalid parameters were used or it cannot lock **HPENDATA.**

# GetPenDataStroke

**Action** Returns a pointer to the data

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL GetPenDataStroke**(*lppendata, wStroke, lplpPoint, lplpvOem, lpsi*)

| Parameter | Type | Description |
|---|---|---|
| *lppendata* | **LPPENDATA** | Pen data to be enumerated. This parameter is the value returned by a previous call to the **BeginEnumStrokes** function. |
| *wStroke* | **UINT** | Which stroke to get. The number of the stroke is zero-based. |
| *lplpPoint* | **LPPOINT FAR *** | Pointer to the data. You should be careful about modifying the data in your application. This pointer becomes invalid when you call **EndEnumStrokes**. |
| *lplpvOem* | **LPVOID FAR *** | OEM data block. If *lplpvOem* is NULL, no OEM data is returned. If it is not NULL, *lplpvOem* points to the OEM data block. The format of the OEM data is specified by the **peninfo** field in the **PENDATAHEADER** structure. |
| *lpsi* | **LPSTROKEINFO** | Contains a pointer to the STROKEINFO structure to be filled. |

**Comments** The **GetPenDataStroke** function returns a pointer in *lplpPoint* to the pen data associated with the stroke at the position specified by *wStroke*. The data is an array of **POINTS**.

You must call **BeginEnumStrokes** before calling **GetPenDataStroke**. After the last call to **GetPenDataStroke,** you must call **EndEnumStrokes**. Once **EndEnumStrokes** is called, the points contained in *lplpPoint* and *lplpvOem* are no longer valid.

**Return Value** **GetPenDataStroke** returns TRUE if the call succeeds. If the stroke requested is out of range, it returns FALSE.

**See Also** **BeginEnumStrokes, EndEnumStrokes**

# GetPenHwData

**Action** Fetches data from the internal pen buffer

**Module** RC Manager

**Called By** Recognizer

**Syntax** **REC GetPenHwData**(*lpPnt, lpvOemData, cPntMax, wTimeOut, lpsi*)

| Parameter | Type | Description |
|---|---|---|
| *lpPnt* | **LPPOINT** | Buffer to fill with pen event data. |
| *lpvOemData* | **LPVOID** | Buffer to fill with OEM-specific data (an array of OEM words). This can be NULL if no data is required. |
| *cPntMax* | **int** | Maximum number of samples to return. |
| *wTimeOut* | **UINT** | **GetPenHwData** returns REC_TIMEOUT if *wTimeOut* msecs have elapsed since the last time pen down data was added to the queue. |
| *lpsi* | **LPSTROKEINFO** | Stroke information, including the count of points and point state. Also included is the time stamp of the first point returned in the buffer. It is expressed in milliseconds since the start of collection mode. |

**Comments** The **GetPenHwData** function fetches data from the internal pen buffer. It should be called only by the recognizer.

The first call to this function returns the data pointed to by the **rc.wEventRef** passed to **Recognize**. Subsequent calls return data immediately following the previously returned data.

**GetPenHwData** returns (in *lpsi)* the stroke information. In addition, **GetPenHwData** returns in *lpPnt* points which represent either all pen-down or all pen-up points. In other words, on a single call, **GetPenHwData** will stop filling the buffer when the next point to add is not in the same state as the previous point. Consecutive calls may return points of the same state if there is not enough room in the return buffer or if the data was not yet available. For more details on the **STROKEINFO** structure, see Chapter 10, "Pen Structures."

The *lpPnt* parameter must be at least **sizeof(POINT)****cPntMax* and *lpvOemData* must be NULL or *cbOemData*cPntMax* in size to avoid overflow. (The *cbOemData* parameter is defined in the **PENINFO** structure using the function). If no data is available, *lpsi->cPnt* == 0 and the return value is REC_OK.

The collection of data ends with a nonzero return value.

**Return Value** **GetPenHwData** returns the following values.

| Value | Meaning |
|---|---|
| REC_OK | The call is successful. The *lpPnt* parameter contains *lpsi.cPnt* points of valid data. The *lpsi.cPnt* parameter may equal zero if no data is currently available. |
| REC_OVERFLOW | The driver buffer has overflowed, with possible loss of data. The *lpPnt* data is not valid. |
| REC_ABORT | Recognition was halted by a call to **EndPenCollection** with this value. The *lpPnt* data is not valid. |
| REC_NOCOLLECTION | Pen collection mode has not been set. |
| REC_PARAMERROR | An invalid parameter has been passed to **GetPenHwData**. |
| REC_TERMBOUND | The call was ended because of a hit test outside the bounding rectangle. The *lpPnt* parameter is filled with the point causing the stop. |
| REC_TERMEX | The call was ended because of a hit test inside the exclusion rectangle. The *lpPnt* parameter is filled with the point causing the stop. |
| REC_TERMPENUP | The call was ended on pen up. The *lpPnt* parameter is filled with the pen-up point that ended the recognition. |
| REC_TERMRANGE | The call was ended because the pen left the proximity range. The *lpPnt* data is not valid. |
| REC_TERMTIMEOUT | The call was ended on time out. |
| REC_TERMOEM | Values >= 512 are reserved for termination reasons specific to the recognizer. |

**See Also** **AddPointsPenData, EndPenCollection**, **STROKEINFO** structure

# GetPenHwEventData

**Action** Gets the pen data associated with events *wEventRefBeg* through *wEventRefEnd*

**Module** RC Manager

**Called By** Application

**Syntax** **REC GetPenHwEventData**(*wEventRefBeg, wEventRefEnd, lpPnt, lpvOemData, cPntMax, lpsi*)

| Parameter | Type | Description |
|---|---|---|
| *wEventRefBeg* | **UINT** | Beginning pen event. |
| *wEventRefEnd* | **UINT** | Ending pen event. |
| *lpPnt* | **LPPOINT** | Buffer to fill to with (x,y) data. |
| *lpvOemData* | **LPVOID** | Buffer to fill with OEM-specific data. This can be NULL if no data is required. |
| *cPntMax* | **int** | Maximum number of samples to return. |
| *lpsi* | **LPSTROKEINFO** | Filled with stroke information including the count of points and point state. Also included is the time stamp of the first point returned in the buffer, recorded in milliseconds since Windows startup. |

**Comments** This function fetches all data collected from the pen event *wEventRefBeg* through the pen event *wEventRefEnd*, excluding the final point. If *wEventRefBeg* equals *wEventRefEnd*, **GetPenHwEventData** fetches the pen event associated with *wEventRefBeg*.

The values for *wEventRefBeg* and *wEventRefEnd* are obtained by a call to **GetMessageExtraInfo**.

Unlike **GetPenHwData,** this function can be called directly from an application.

**Return Value** The following table lists the values returned by **GetPenHwEventData**.

| Value | Meaning |
|---|---|
| REC_OK | The buffer was successfully filled. |
| REC_BUFFERTOOSMALL | The *cPntMax* buffer is not large enough. |
| REC_PARAMERROR | An invalid parameter has been passed to **GetPenHwEventData** |

If REC_BUFFERTOOSMALL is returned, no data is returned and the **cPnt** field of *lpsi*

contains the number of points between *wEventRefBeg* and *wEventRefEnd*. If REC_OK is returned, *lpcPnt* contains the number of valid points placed in *lpPnt*.

**See Also** **GetPenHwData, GetMessageExtraInfo**.

# GetPointsFromPenData

**Action** Gets the specified data points

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL GetPointsFromPenData**(*hpendata, wStroke, wPnt, cPnt, lppoint*)

| Parameter | Type | Description |
|---|---|---|
| *hpendata* | **HPENDATA** | Handle to pen data structure |
| *wStroke* | **UINT** | Which stroke to get |
| *wPnt* | **UINT** | Beginning point in *wStroke* |
| *cPnt* | **UINT** | Number of points to get |
| *lppoint* | **LPPOINT** | Buffer to fill |

**Comments** The **GetPointsFromPenData** function copies the points between *wPnt* and *wPnt* + *cPnt* in the stroke specified by *wStroke* to the buffer *lppoint*.

If *wStroke* is greater than the number of strokes in *hppendata*, **GetPointsFromPenData** returns the points from the last stroke. If the count of points to return is 1 and *wPnt* is beyond the last point in the stroke, **GetPointsFromPenData** returns the last point in the stroke.

**Return Value** **GetPointsFromPenData** returns TRUE if the call succeeds. If you are asking for points that are out of range, it returns FALSE.

**See Also** **GetPenDataStroke**

# GetSymbolCount

**Action** Returns the number of symbol string enumerations possible in a symbol graph

**Module** RC Manager

**Called By** Application

**Syntax** **int GetSymbolCount**(*lpsyg*)

| Parameter | Type | Description |
|---|---|---|
| *lpsyg* | **LPSYG** | Pointer to symbol graph |

**Return Value** This function 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.

For example, if the symbol graph *lpsyg* is `ex { a | u } mple`, a call to **GetSymbolCount** returns the value 2, because two symbol strings can be generated ("example" and "exumple").

**See Also** **EnumSymbols, FirstSymbolFromGraph, GetSymbolMaxLength**

# GetSymbolMaxLength

**Action** Gets the length of the longest symbol string

**Module** RC Manager

**Called By** Application

**Syntax** **int GetSymbolMaxLength**(*lpsyg*)

| Parameter | Type | Description |
|---|---|---|
| *lpsyg* | **LPSYG** | Pointer to the symbol graph |

**Return Value** This function returns the number of symbols in the longest symbol string that can be generated from the symbol graph. For example, if the symbol graph *lpsyg* is `ab {c | de } f`, a call to **GetSymbolMaxLength** returns 5, because the longest string is "abdef."

**See Also** **EnumSymbols, FirstSymbolFromGraph**

# GetVersionPenWin

**Action** Gets the version number of Microsoft Windows for Pen Computing

**Module** RC Manager

**Called By** Application

**Syntax** **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.

# InitRC

**Action** Fills an **RC** structure with the default values

**Module** RC Manager

**Called By** Application

**Syntax** **void InitRC**(*hwnd, lprc*)

| Parameter | Type | Description |
|---|---|---|
| *hwnd* | **HWND** | Window handle |
| *lprc* | **LPRC** | Pointer to **RC** structure |

**Comments** **InitRC** fills an **RC** structure with the default values.The resulting **RC** structure is a valid **RC** structure that can be passed to **Recognize.** 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 Control Panel.

**InitRC** sets the bounding rectangle to the client area of *hwnd*. The bounding rectangle set by **InitRC** is valid only until the window is resized or moved, when it becomes invalid. Therefore, an application cannot use **InitRC** once and then use the **rectBound** field in the resulting **RC** without modification. If the window handle *hwnd* is NULL, then the bounding rectangle and **rc.hwnd** are left unitialized. The **rc.hwnd** field must be set to a valid window before **Recognize** or **RecognizeData** is called.

Values not listed in the following table are set to the value in the global **RC** structure. Some of the global default values can be modified by the user in the Control Panel. These global values are described in Chapter 11, "Pen Messages and Constants."

| RC structure field | Value |
|---|---|
| **rc.alc** | ALC_DEFAULT. That is, the function uses the complete alphabet and all gestures. The exact character set is recognizer-dependent. |
| **rc.lRcOptions** | 0 |
| **rc.hwnd** | *hwnd* (the argument) |
| **rc.wResultMode** | RRM_COMPLETE |
| **rc.rectBound** | (0,0,0,0) or client rectangle of *hwnd* if *hwnd* is not NULL |
| **rc.lPcm** | PCM_ADDDEFAULTS |
| **rc.rectExclude** | (0,0,0,0) |
| **rc.guide** | (0,0,0,0,0,0,0) |
| **rc.wRcOrient** | RCOR_NORMAL |
| **rc.wRcDirect** | RCD_DEFAULT |

Those values that can be changed 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 field 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 **rc.lPcm**, the values of the **lPcm** field in the global **RC** are combined with the current **rc.lPcm** values with OR operators at the time the recognizer is called.

If the high bit is set in **rc.wRcPreferences**, the values of the **wRcPreferences** field in the global RC are combined with the current **rc.wRcPreferences** values with OR operators at the time the recognizer is called.

| RC structure field | Default value |
|---|---|
| **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** | 100 |
| **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 zero |

The **RC** structure pointed to in the **RCRESULTS** structure is a copy of the original **RC** structure passed in as a parameter to **Recognize**. In this copy, the default values are replaced; all coordinates are in tablet coordinates, and the **lRcOptions** field have the RCO_TABLETCOORD flag set.

**Return Value** None

**See Also** **Recognize, RecognizeData**

# InitRecognizer

**Action** Gives the recognizer a chance to do any initialization before receiving the first request for recognition

**Module** Recognizer

**Called By** RC Manager

**Syntax** **BOOL InitRecognizer**(*lprc*)

| Parameter | Type | Description |
|---|---|---|
| *lprc* | **LPRC** | Long pointer to **RC** structure |

**Comments** **InitRecognizer** is called when the recognizer is loaded by a call to **InstallRecognizer** after the DLL's **LibMain** but before any other function in the recognizer's DLL. It is called with a copy of the global **RC**.

Applications should not call this function directly but should call **InstallRecognizer** instead.

**Return Value** Returns FALSE if the load fails. The recognition load can fail if the recognizer requires resources that are not available—for example, a pressure-sensitive pen. It is the recognizer's responsibility to post any error messages on failure.

**See Also** **CloseRecognizer, InstallRecognizer, SetGlobalRC**

# InstallRecognizer

**Action** Loads a specified recognizer

**Module** RC Manager

**Called By** Application

**Syntax** **HREC InstallRecognizer**(*lpszRecogName*)

| Parameter | Type | Description |
|---|---|---|
| *lpszRecogName* | **LPSTR** | Recognizer module name to load. If *lpszRecogName* is NULL, the default recognizer is loaded. Windows does this automatically on initialization. |

**Comments** The recognizer's name is the name of the DLL to be loaded. The standard rules for searching for a DLL are used. The procedure fails if the library cannot be found, the load fails, or the loaded DLL is not a valid recognizer. The load may also fail if the recognizer requires services that are not provided by the current pen driver—for example, pressure.

After the recognizer is loaded, **InitRecognizer** is called by the RC Manager. 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.

If an application loads a recognizer with a call to **LoadLibrary** instead of **InstallRecognizer**, only the **ConfigRecognizer** function in the recognizer can be called.

**Return Value** The return value is a handle to a recognizer to be used in the **RC** structure. **InstallRecognizer** returns NULL on an error.

**See Also** **InitRecognizer, UninstallRecognizer**

# IsPenAware

**Action** Checks the capability of an application to handle pen events

**Module** RC Manager

**Called By** System Application

**Syntax** **UINT IsPenAware**()

**Comments** This function is called to determine which handwriting events are handled by the application for the current task. The information returned is the registration flags previously set by a call to the **RegisterPenApp** function.

**Return Value** **IsPenAware** returns the registration flags word set by a previous call to the **RegisterPenApp** function. If **RegisterPenApp** has not been called previously, **IsPenAware** returns zero.

**See Also** **RegisterPenApp**

# IsPenEvent

**Action** Checks to see if a mouse event was generated by the pen driver

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL IsPenEvent**(*message*, *lExtraInfo*)

| Parameter | Type | Description |
|---|---|---|
| *message* | **UINT** | Windows mouse message being queried |
| *lExtraInfo* | **LONG** | Value returned by **GetMessageExtraInfo** for the given message |

**Note** 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.

**Return Value** This function returns TRUE if the given mouse event was generated by the pen driver. All other messages return FALSE.

# MetricScalePenData

**Action** Converts pen data points to one of the supported metric modes

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL MetricScalePenData**(*hpendata, wPdts*)

| Parameter | Type | Description |
|---|---|---|
| *hpendata* | **HPENDATA** | Pen data points to be converted |
| *wPdts* | **UINT** | Scaling metric to be used with the data |

**Comments** The **MetricScalePenData** function scales the points in *hpendata* according to the scaling values given by the *wPdts* parameter.

The following table lists the *wPdts* scaling values.

| Scaling value | Meaning |
|---|---|
| PDTS_LOMETRIC | Each logical unit is mapped to 0.1 mm. Positive x is to the right; positive y is down. |
| PDTS_HIMETRIC | Each logical unit is mapped to 0.01 mm. 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 will be left in display coordinates. Strictly speaking, this is not a metric scale. |

To use the PDTS_DISPLAY scale type, the current scale of the data must be in PDTS_STANDARDSCALE units. The PDTS_DISPLAY scale also assumes that the desired tablet-to-display ratio is equal to the current ratio for the display and tablet.

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** field of the *pendata* 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, the conversion of scalings is not perfectly reversible.

All recognizers must recognize points that have been scaled to PDTS_STANDARDSCALE (equivalent to PDTS_HIENGLISH).

**Return Value** **MetricScalePenData** returns TRUE if successful; it returns FALSE if *hpendata* is in a compressed state or if the data is not already in one of the metric modes—for example, if the data is in PDTS_ARBITRARY mode.

**See Also** **OffsetPenData, ResizePenData**

# OffsetPenData

**Action** Offsets pen data points by a specified amount

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL OffsetPenData**(*hpendata, dx, dy*)

| Parameter | Type | Description |
|---|---|---|
| *hpendata* | **HPENDATA** | Handle to pen data memory block. |
| *dx* | **int** | x-axis offset: amount to move left or right. To move left, the *dx* value must be negative. |
| *dy* | **int** | y-axis offset: amount to move up or down. To move up, the *dy* value must be negative. |

**Comments** For every point in hpendata, *dx* is added to the x-coordinate and *dy* is added to the y-coordinate. No overflow checks are made.

**Return Value** **OffsetPenData** returns TRUE if successful, FALSE if *hpendata* is in a compressed state.

**See Also** **MetricScalePenData, ResizePenData**

# PostVirtualKeyEvent

**Action** Posts a virtual key code event to Windows

**Module** RC Manager

**Called By** System applications (Pen Palette)

**Syntax** **void PostVirtualKeyEvent**(*vk*, *fUp*)

| Parameter | Type | Description |
|---|---|---|
| *vk* | **UINT** | Virtual key |
| *fUp* | **BOOL** | Key transition—FALSE for down, TRUE for up |

**Comments** This function does not check the virtual key code for errors.

You can post repeating keys by calling **PostVirtualKeyEvent** consecutively with *fUp* FALSE. End this by a single call with *fUp* set to TRUE.

The events are posted to the system message queue and can be received by the application with the input focus, using the **GetMessage** and **PeekMessage** calls.

**Return Value** None

**See Also** **AtomicVirtualEvent, PostVirtualMouseEvent**

# PostVirtualMouseEvent

**Action** Posts a virtual mouse code event to Windows

**Module** RC Manager

**Called By** System applications (Pen Palette)

**Syntax** **void PostVirtualMouseEvent**(*wMouseFlag*, *xPos*, *yPos*)

| Parameter | Type | Description |
|---|---|---|
| *wMouseFlag* | **UINT** | Mouse flag |
| *xPos* | **int** | The x position in screen coordinates |
| *yPos* | **int** | The y position in screen coordinates |

**Comments** The x and y positions are absolute positions in screen coordinates. Note that the *x* and *y* values should not exceed the screen resolution limits. The *wMouseFlag* parameter is one of the values in the following table.

| Constant | Value |
|---|---|
| VWM_MOUSEMOVE | 0x0001 |
| VWM_MOUSELEFTDOWN | 0x0002 |
| VWM_MOUSELEFTUP | 0x0004 |
| VWM_MOUSERIGHTDOWN | 0x0008 |
| VWM_MOUSERIGHTUP | 0x0010 |

The VWM_MOUSELEFTDOWN and VWM_MOUSELEFTUP constants are set only to mark a transition in state. The VWM_MOUSEMOVE constant is set if the cursor changes positions. A move and button transition can be entered in the same event by connecting the correct flags with OR operators.

The events are posted to the system message queue and can be received by the application with the input focus, using the **GetMessage** and **PeekMessage** calls.

Because of the way Windows interprets mouse messages, the caller 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 will generate first a Windows event for the button transition at the current cursor location and then a move to the new location. If the expected result is a move to a new location and then a button transition, this requires two separate calls to **PostVirtualMouseEvent**.

When posting events, the caller should bracket the calls by calls to **AtomicVirtualEvent,** which is used to lock 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 );
```

**GetMessageExtraInfo** will return 0 for any messages generated with this function.

**Return Value** None

**See Also** **AtomicVirtualEvent, GetMessageExtraInfo, PostVirtualKeyEvent**

# ProcessPenEvent

**Action** Tells the pen module to process any pending pen events

**Module** RC Manager

**Called By** Pen driver

**Syntax**

```
call    dword ptr lpfnProcessPenEvent
```

**Comments** The **ProcessPenEvent** function tells the pen module to process any queued pen events that have been added with the **AddPenEvent** function. This function is reentrant and can be called with all interrupts—including tablet interrupts—enabled.

Processing a pen packet may involve graphics operations, copying large amounts of memory, calling numerous Windows APIs, and other time-consuming operations. Therefore, it is recommended that a pen driver enable the tablet interrupts before the call to **ProcessPenEvent.**

A pen driver should construct a pen packet with tablet interrupts disabled, call **AddPenEvent**, re-enable tablet interrupts, and then call **ProcessPenEvent**.

**Return Value** None

**See Also** **AddPenEvent**

# ProcessWriting

**Action** Processes handwriting

**Module** RC Manager

**Called By** Application

**Syntax** **REC ProcessWriting**(*hwnd, lprc*)

| Parameter | Type | Description |
|---|---|---|
| *hwnd* | **HWND** | Window to receive messages. This parameter must not be NULL. |
| *lprc* | **LPRC** | Pointer to **RC** structure to use for recognition. This parameter can be NULL. |

**Comments** The **ProcessWriting** function simplifies the task of converting an existing application to take advantage of handwriting input—both gestures and characters. This function is similar to **Recognize** except that, in addition, **ProcessWriting** 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.

The *hwnd* parameter is the window to receive messages. It cannot be NULL. The *lprc* parameter is the **RC** structure to use for recognition. If the *lprc* parameter is an **RC,** the results mode field is ignored and the recognition terminates on RRM_COMPLETE.

The window specified by the *hwnd* parameter receives a WM_PARENTNOTIFY message when **ProcessWriting** destroys its inking window. The *wParam* value is WM_DESTROY, and *lParam* contains –1 in the HIWORD. The LOWORD of *lParam* contains the window handle of the inking window being destroyed. An application may ignore this message.

If *lprc* is NULL, a default **RC** structure is created for the application. The default **RC** structure contains all system defaults. In addition, the inking is constrained to the client area of *hwnd*. Otherwise, *lprc->rectBound* is used to constrain the inking.

The **ProcessWriting** function is normally called on the mouse-click message WM_LBUTTONDOWN or at other times with a non-NULL *lprc* that contains a valid *wEventRef.*

On a call to **Recognize** or **ProcessWriting,** during a WM_LBUTTONDOWN message, the application does not receive the corresponding pen up message.

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 a nonzero value. If the application returns 0 (**DefWindowProc** will return 0 for this message), **ProcessWriting** performs the default conversion of the results message to standard Windows messages.

By default, when an application receives a WM_RCRESULT message as a result of a **ProcessWriting** call, the *lprcresult->hpendata* value is NULL. If you want to have the *hpendata* returned, set the RCO_SAVEHPENDATA flag in the *rc.lRcOptions* field. If this is done, the caller must free *hpendata.*

If an application returns FALSE to the WM_RCRESULT message, the application receives the Windows messages shown in the following table. The messages are sent rather than posted. If the application returns TRUE to the WM_RCRESULT message, no further messages are sent.

| Result | Messages to hwnd |
|---|---|
| SYV_BACKSPACE | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture, followed by WM_CHAR of 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 |
| SYV_CORRECT | WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_LBUTTONUP at the hotspot of the gesture, followed by WM_COPY; then the Edit Text dialog box is activated, and it pulls text from the Clipboard. This uses the existing selection if any is present.<br>The previous contents of the Clipboard are lost. |
| SYV_CUT | WM_CUT |
| SYV_EXTENDSELECT | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture. The MK_SHIFT flag is set for the *wParam* of these messages. |
| SYV_PASTE | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture. WM_PASTE. |
| SYV_RETURN | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture, followed by WM_CHAR of RETURN. |
| SYV_SPACE | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture, followed by WM_CHAR of SPACE. |
| SYV_TAB | WM_LBUTTONDOWN, followed by WM_LBUTTONUP at the hotspot of the gesture, followed by WM_CHAR of TAB. |

| Result | Messages to hwnd |
|---|---|
| SYV_UNDO | WM_UNDO |
| text | One WM_CHAR message per character of text. |
| all other results | No messages. |

Additional information is included in Chapter 6, "Using Pen Controls and the **ProcessWriting** Function."

**Return Value** The **ProcessWriting** function returns values of less than 0 if the application should treat the event as a mouse event instead of a pen event. Return values of less than zero occur if the event did not come from a pen, the user performed a press-and-hold action (REC_POINTEREVENT), or an error—for example, running out of memory occurred.

**See Also** **InitRC, Recognize, RC** structure, **RCRESULT** structure

# Recognize

**Action** Begins sampling pen data and converts tablet input to recognized symbols

**Module** RC Manager

**Called By** Application

**Syntax** **REC Recognize**(*lprc*)

| Parameter | Type | Description |
|---|---|---|
| *lprc* | **LPRC** | Recognition parameters |

**Comments** This is the primary recognition function. An application calls **Recognize** to begin recognition on a pen event identified by *lprc ->wEventRef.* (For information on the *wEventRef* parameter, see the entry for **GetMessageExtraInfo,** earlier in this chapter.) The *lprc* parameter contains the parameters that control the recognition. Results are sent through the WM_RCRESULT message to the window indicated in *lprc*. All results messages are sent before this function returns. Multiple result messages may be sent if the application asks for results to be sent to the application before all input has been completed (as indicated by **rc.wResultMode**).

**Recognize** should be called in response to a WM_LBUTTONDOWN message.

On a call to **Recognize** or **ProcessWriting,** the application will not receive the corresponding pen up message.

**Return Value** The following table lists the possible return values. **Recognize** may also return any of the return values from **GetPenHwData**.

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 zero 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 of the values listed below 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** will be the return value of **Recognize**. Some error conditions, such as REC_OOM or REC_NOTABLET, will be returned without any WM_RCRESULT message being generated.

| Value | Meaning |
|---|---|
| REC_OK | 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. |

| Value | Meaning |
|---|---|
| REC_BADHPENDATA | Returned if *hpendata* in *lprc* cannot be locked or has an invalid header. It 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_NOINPUT | Returned by **RecognizeData** if the buffer contains no data; returned by **Recognize** if recognition ends before any data is collected, for example, a pen down occurs outside the bounding rectangle before any data is 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 holds the position for a short period of time. This return value indicates that the application should begin selection actions rather than inking or recognition. If this is returned, no WM_RCRESULT message is generated and no ink will be displayed. |
| REC_TERMBOUND | Terminated because of a hit test outside the bounding rectangle. The **rcresult.pntEnd** field is filled with the point causing the stop. |
| REC_TERMEX | Terminated because of a hit test inside the exclusion rectangle. The **rcresult.pntEnd** field is filled with the point causing the stop. |
| REC_TERMOEM | Values >= reserved for recognizer-specific termination reasons. |
| REC_TERMPENUP | Terminated on pen up. The **rcresult.pntEnd** field is filled with the pen-up point that terminated recognition. |
| REC_TERMRANGE | Terminated because the pen left the proximity range. |
| REC_TERMTIMEOUT | Terminated on time-out. (The pen was up continuously for a given amount of time.) |

## Debugging Values

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**.

| Value | Meaning |
|---|---|
| REC_DEBUG | All debugging return values are less than this. |
| REC_ALC | Invalid enabled alphabet. |
| REC_BADEVENTREF | Returned when the *wEventRef* field in the *lprc* 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 | This value is returned by the recognizer when the **lpLanguage** field contains a language that is not supported by the recognizer. Call **ConfigRecognizer** withWCR_QUERYLANGUAGE subfunction to determine whether a particular language is supported. |
| REC_NOCOLLECTION | Returned by **GetPenHwData** if collection mode has not been set. |
| REC_OEM | Error values below this (below –1024) are specific to the recognizer. |
| REC_PCM | Invalid *lPcm* parameter. 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, RC** structure

# RecognizeData

**Action** Converts the data in *hpendata* to recognized symbols

**Module** RC Manager

**Called By** Application

**Syntax** **REC RecognizeData**(*lprc*, *hpendata*)

| Parameter | Type | Description |
|---|---|---|
| *lprc* | **LPRC** | Pointer to **RC** structure |
| *hpendata* | **HPENDATA** | Buffer containing pen data memory |

**Comments** This function recognizes data in *hpendata* and returns the results to *lprc->hwnd*. It is similar to **Recognize;** the difference is that the input data comes from the parameter instead of the tablet driver. Parameters regarding 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 *hpendata* 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 on to the recognizer. A recognizer may return an error code (REC_BADHPENDATA) on data in a scale it cannot handle or attempt to recognize.

**Return Value** **RecognizeData** returns REC_DONE on completion, or an error code if an error occurs. The error codes are identical to those for **Recognize**. REC_BADHPENDATA is also returned on compressed HPENDATA.

**See Also** **InitRC, Recognize**

# RecognizeDataInternal

**Action** At the DLL recognition level, converts *hpendata* to recognized symbols

**Module** Recognizer

**Called By** RC Manager

**Syntax** **REC RecognizeDataInternal**(*lprc, hpendata, lpFuncResults*)

| Parameter | Type | Description |
|---|---|---|
| *lprc* | **LPRC** | Recognition parameters |
| *hpendata* | **HPENDATA** | Pen data structure |
| *lpFuncResults* | **FARPROC** | Pointer to function |

**Comments** This function is similar to **RecognizeInternal;** the difference is that data for recognition is provided as a parameter instead of being read from the pen driver. This function is called by an application's call to **RecognizeData**.

**Return Value** Same as for **Recognize**.

**See Also** **Recognize**, **RecognizeInternal**

# RecognizeInternal

**Action** Begins sampling pen data and converts tablet input to recognized symbols

**Module** Recognizer

**Called By** RC Manager

**Syntax** **REC RecognizeInternal**(*lprc*, *lpFuncResults*)

| Parameter | Type | Description |
|---|---|---|
| *lprc* | **LPRC** | Recognition parameters |
| *lpFuncResults* | **FARPROC** | Pointer to function |

**Comments** This function is called in the DLL when an application calls **Recognize**. The **RecognizeInternal** function is in the recognizer and cannot be called directly by an application.

**RecognizeInternal** calls (**lpFuncResults*)() with a result in an **RCRESULT** structure. The rules governing which results to send from the recognizer to the RC Manager and when to send them are the same as the rules that govern the sending of results from the RC Manager to the application. This procedure does not return until (**lpFuncResults*)() has been called for the last time.

If lprcresult->hSyv is NULL, the RC Manager assumes that the recognizer has already performed dictionary processing; otherwise, the RC Manager takes the responsibility. The recognizer should set the field lprcresult->hSyv to NULL every time to indicate that it has peformed dictionary processing. The RC Manager is responsible for freeing hSyv. It changes lprcresult so that a recognizer will reset all fields on each call to lpFuncResults to the appropriate values. The RC Manager modifies the wResultType, cSyv, and hSyv fields.

## lpFuncResults

**int (* lpfnFuncResults)**(*lprcresult, wParam*)

| Parameter | Type | Description |
|---|---|---|
| *lprcresult* | **LPRCRESULT** | This is a pointer to the **RCRESULTS** structure. |
| *wParam* | **UINT** | The *wParam* parameter has the same meaning as *wParam* in the WM_RCRESULT message. |

The following table lists the return values for **lpFuncResults.**

| Return value | Meaning |
|---|---|
| 0 | Continues default processing. |
| 1 | Message processed; further processing on this message stopped (for **ProcessWriting**). |
| REC_ABORT | Aborts recognition. If RRM is set for multiple messages per recognizer call, this stops further recognition attempts. It is not applicable to **ProcessWriting**. |
| <0 | Aborts recognition and returns the value from **Recognize.** |

**Return Value** Same as **Recognize**

**See Also** **RecognizeDataInternal**

# RedisplayPenData

**Action** Redraws the pen data in the same manner as originally inked

**Module** RC Manager

**Called By** RC Manager, application

**Syntax** **BOOL RedisplayPenData**(*hDC, hpendata, lpDelta lpExt, nInkWidth, rgbColor*)

| Parameter | Type | Description |
|---|---|---|
| *hDC* | **HDC** | Handle to the device context in which to draw the ink. The mapping mode is assumed to be MM_TEXT. |
| *hpendata* | **HPENDATA** | Handle to the pen data memory block to be displayed. The pen data must be scaled to PDTS_STANDARDSCALE or PDTS_DISPLAY. |
| *lpDelta* | **LPPOINT** | 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* | **LPPOINT** | Extents in logical units for scaling. If *lpExt* is NULL, no scaling is performed. |
| *nInkWidth* | **int** | Width of the ink to be drawn (1–15).<br>If *nInkWidth* is –1, the ink width specified in the *nInkWidth* field in *hpendata* is used for drawing. |
| *rgbColor* | **DWORD** | RGB value of the color to draw the ink. If *rgbColor* is 0xFFFFFFFF, the ink color specified in the *rgbInk* field in *hpendata* is used for drawing. |

**Comments** **RedisplayPenData** displays the data specified by *hpendata* in the device context specified by the *hDC* parameter with a width specified by *nInkWidth* and of color *rgbColor*. The *nInkWidth* and *rgbColor* values override the pen currently selected for the *hDC*.

If the mapping mode of *hDC* 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 prescaling is done, you will have rounding errors when converting between modes. The ink will still shift slightly when repainted.

For any rendering into an *hDC* that represents anything other than a display *hDC*, **DrawPenData** should be used. This is so because **RedisplayPenData** makes assumptions that are not optimal for other devices such as printers or metafiles. **RedisplayPenData** provides the ability to recreate original inking perfectly.

To recreate inking perfectly, an application must follow one of two procedures:

- On receipt of a WM_RCRESULT message, convert the *hpendata* to display coordinates using **MetricScalePenData**; offset the pen data by the display coordinates of the window containing the ink. To display, call **RedisplayPenData** with *lpDelta* and *lpExt* set to NULL and the mapping mode of the *hDC* set to MM_TEXT.
- On receipt of a WM_RCRESULT message, the application must remember the display coordinate of the upper-left corner of the client area of the window containing the ink. To display, call **RedisplayPenData** with *lpDelta* set to this saved coordinate; *lpExt* set to NULL, and the mapping mode of the *hDC* set to MM_TEXT.

For more details on inking, see Chapter 4, "Managing Ink in Pen Applications."

The *lpDelta* parameter contains offsets in the current mapping mode of the device context *hDC* that should be subtraced from all ink coordinates before they are rendered.

Since the pen data has the origin of (0,0) based on the upper-left corner of the display, you need to move from a screen-relative position to a device-context-relative position. Subtract the origin of the device context in screen coordinates from the object currently residing in screen coordinate space.

The *lpDelta* argument enables the application to render ink in a window-relative manner instead of a screen-relative manner. An application should call the Windows function **ClientToScreen** for (0,0) to find the proper screen coordinates to be placed in the ***lpOrg POINT** structure. Once this is done, the pen data will be 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 will be 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 to which the data will be rendered.

**Return Value** The **RedisplayPenData** function returns TRUE if succesful, FALSE if it fails because of invalid parameters.

**See Also** **DrawPenData**

# RegisterPenApp

**Action** Used by pen-enhanced applications to notify the RC Manager that the application edit controls are replaced with hedit controls

**Module** RC Manager

**Called By** Application

**Syntax** **void RegisterPenApp**(*wFlags, fRegister*)

| Parameter | Type | Description |
|---|---|---|
| *wFlags* | **UINT** | Registration flags for recognizer to determine whether or not application is pen-enhanced. |
| *fRegister* | **BOOL** | Sets TRUE to register pen-enhanced application. To unregister the application, this value is set to FALSE. |

**Comments** The **RegisterPenApp** function makes it possible to replace all edit controls in an application with hedits. This simplifies the tasks of making an application pen-aware and making that same application run under both Windows for Pen Computing and regular Windows.

The only registration flag currently supported is RPA_DEFAULT. Other values are reserved for future use. The RC Manager will use the **IsPenAware** function to determine the flags set.

The application should call **RegisterPenApp** before any edit controls or combo dropdown controls are created. The function should also be called with *fRegister* set to FALSE before the application ends.

**Return Value** None

**See Also** **IsPenAware**

# ResizePenData

**Action** Resizes pen data

**Module** Recognizer

**Called By** Application

**Syntax** **BOOL ResizePenData**(*hpendata, lprect*)

| Parameter | Type | Description |
|---|---|---|
| *hpendata* | **HPENDATA** | Handle to pen data memory block |
| *lprect* | **LPRECT** | Bounding rectangle |

**Comments** This function is used to change 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 *hpendata* to the bounding rectangle dimensions given by the *lprect* parameter. Data from *hpendata* is mapped to the new rectangle. If *lprect* is NULL, this function just recalculates the bounding rectangle (the **rectBound** element in the PENDATAHEADER structure).

For example, assume that the current object is a square and is at location (1500, 1600) with PDTS_HIMETRIC scaling, and you need to double the size. To accomplish this, set *lprect* to (500, 600, 2500, 2600).

**Return Value** **ResizePenData** returns TRUE if successful.

**See Also** **OffsetPenData, MetricScalePenData**

# SetGlobalRC

**Action** Sets the current default settings for the **RC** structure

**Module** RC Manager

**Called By** System applications (Pen Palette, Control Panel, and others)

**Syntax** **UINT SetGlobalRC**(*lprc*, *lpDefRecog*, *lpDefDict*)

| Parameter | Type | Description |
|---|---|---|
| *lprc* | **LPRC** | Pointer to **RC** structure. |
| *lpDefRecog* | **LPSTR** | Default recognizer module name (maximum 128 bytes). |
| *lpDefDict* | **LPSTR** | Default dictionary path. The list should end with double zero bytes. |

**Comment** Because the default **RC** values are shared among all applications on the system, they should be changed only through the Control Panel. Whenever a change is made to the global **RC** values, the WM_GLOBALRCCHANGE message is sent to all top-level windows. The *wParam* and *lParam* values are not used; they are set to zero.

Any of the parameters may be NULL to indicate that the caller does not want the value changed.

The standard rules for searching a DLL are used for the recognizer and each one of the dictionaries.

**SetGlobalRC** uses only certain fields of the RC structure passed in through the *lprc* parameter. They are the following:

| | |
|---|---|
| **wRcPreferences** | **lpUser** |
| **wCountry** | **wIntlPreferences** |
| **lpLanguage** | **wTryDictionary** |
| **clErrorLevel** | **wTimeOut** |
| **rgbInk** | **wRcDirect** |
| **nInkWidth** | the PCM_TIMEOUT and PCM_RANGE bits of **lPcm** |

When **InitRecognizer** is called for a new recognizer from within the **SetGlobalRC** call, the **RC** structure that is passed in contains the new values for all fields except **hrec** and **rglpdf;** no new recognizer and dictionaries have been set up at this point.

When an application receives a WM_GLOBALRCCHANGE 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 will call the **ConfigRecognizer** function in the new default recognizer with a CR_RCCHANGE subfunction request.

In a similar way, an application should also call **DictionaryProc** with a DIRQ_RCCHANGE subfunction request for any dictionaries it has loaded, excluding dictionaries on the default path. The RC Manager will call the **DictionaryProc** function in the new default dictionaries with a DIRQ_RCCHANGE subfunction request.

**SetGlobalRC** does not save the RCP_MAPCHAR flag in the **wRcPreferences** field of the **RC** structure to the PENWIN.INI file. The RCP_MAPCHAR flag is reflected in the global **RC** for the current session only.

**Return Value** **SetGlobalRC** returns the value SGRC_OK if successful. If an error occurs, the return value consists of one or more of the other SGRC_ flags, combined with the bitwise OR operation.

The following table lists the SGRC_ values.

| Value | Meaning |
|---|---|
| SGRC_OK | There are no errors. No other flags are set. |
| SGRC_USER | An invalid user name was found in the supplied RC structure *lprc*. 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, *DefRecog*is invalid; or the recognizer cannot be loaded. The supplied recognizer module name is ignored. |
| SGRC_DICTIONARY | The supplied dictionary path, *lpDefDict*, 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 initialization file PENWIN.INI. The new settings will be lost after rebooting Windows. |

**See Also** **GetGlobalRC**

# SetPenHook

**Action** Installs and removes a pen packet hook

**Module** RC Manager

**Called By** System applications (Pen Palette)

**Syntax** **BOOL SetPenHook**(*hkpOp, lpfn*)

| Parameter | Type | Description |
|---|---|---|
| *hkpOp* | **HKP** | Operation to be performed |
| *lpfn* | **LPFNRAWHOOK** | Function to handle pen packets |

**Comments** This function is called by any routine that needs to examine, modify, or cancel pen packets as they arrive.

The operation parameter, *hkpOp*, determines whether the hook is set or removed. The following table lists the HKP_ values.

| Value | Meaning |
|---|---|
| HKP_SETHOOK | Installs a hook |
| HKP_UNHOOK | Removes function from hook list |

**Return Value** The function returns FALSE if it is unable to set or remove the hook.

**PenHookCallBack** is a callback function that examines, modifies, or cancels a pen packet. Each callback is run at interrupt time on every pen packet.

## PenHookCallBack

**BOOL lpfn PenHookCallBack**(*lppp*)

| Parameter | Type | Description |
|---|---|---|
| *lppp* | **LPPENPACKET** | Pointer to the pen packet that is being processed |

**Return Value** The callback function returns FALSE to cancel the processing of this pen packet.

**See Also** **SetRecogHook**

# SetRecogHook

**Action** Installs and removes recognition hook

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL SetRecogHook**(*whrHook, hkpPosition, hwndHook*)

| Parameter | Type | Description |
|---|---|---|
| *whrHook* | **UINT** | Hook parameter |
| *hkpOp* | **UINT** | Operation to be performed |
| *hwndHook* | **HWND** | Handle to a window |

**Comments** This function enables an application to examine the results of recognition before they are sent to the target application.

The operation parameter, *hkpOp*, determines whether the hook is set or removed. The following table lists the HKP_ values.

| Value | Meaning |
|---|---|
| HKP_SETHOOK | Installs a hook |
| HKP_UNHOOK | Removes function from hook list |

The hook parameter, *whrHook*, determines the scope of the hook. The following table lists the HWR_ values.

| Value | Meaning |
|---|---|
| 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 a WM_HOOKRCRESULT message 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 HForm sample application demonstrates a typical use of this feature.<br><br>The RCRT_ALREADYPROCESSED flag is set in the **wResultsType** field of the results structure sent with WM_RCRESULT if an application-wide hook has already processed the data. |

The hook message is WM_HOOKRCRESULT. The *wParam* and *lParam* parameters are the same as for the WM_RCRESULT message.

If the windows procedure that receives the the WM_HOOKRESULT message returns FALSE, the WM_HOOKRESULT message will not be sent to any of the remaining hooks in the chain.

**Note**

No drawing should occur during the processing of the WM_HOOKRESULT and before recognition is complete. Drawing at these times could cause timing problems, with ink reappearing in formerly invisible controls as they are redrawn.

**Return Value**

The function returns FALSE if it is unable to set or remove the hook.

**See Also**

**SetPenHook**

# ShowKeyboard

**Action** Displays or hides the on-screen keyboard

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL ShowKeyboard**(*hwnd, wCommand, lpPnt, lpSKBInfo*)

| Parameter | Type | Description |
|---|---|---|
| *hwnd* | **HWND** | Handle to window invoking the on-screen keyboard. |
| *wCommand* | **UINT** | Command request. |
| *lpPnt* | **LPPOINT** | Pointer to initial keyboard position in screen coordinates or NULL. |
| *lpSKBInfo* | **LPSKBINFO** | Pointer to SKBINFO structure to be filled with values for current keyboard. This parameter is ignored if NULL. If *lpSKBInfo->hwnd* is NULL, no onscreen keyboard has been loaded yet. |

**Comments** The function displays or hides the on-screen keyboard. The *wCommand* argument specifies the action to be taken.

| Command Request | Meaning |
|---|---|
| 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 onscreen keyboard if it is not already present. |
| SKB_QUERY | Returns the current state of the keyboard in *lpSKBInfo* 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 similar to that used by WinHelp. SKB_SHOW automatically loads the on-screen keyboard if it is not present. |

The SKB_SHOW command can be connected by an OR operator with any of the command or keypad requests listed in the two tables below.

| Command request used with SKB_SHOW | Meaning |
|---|---|
| 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 *lpPnt.* If *lpPnt* is NULL, the keyboard will be centered instead. If it is not NULL, *lpPnt* specifies a pointer to the x and y screen coordinates of the upper-left corner of the restored keyboard. |

| Keypad request used with SKB_SHOW | Meaning |
|---|---|
| SKB_BASIC | Switches keyboard to partial keyboard with no extended keys. |
| SKB_FULL | Switches keyboard to full 101-key display. |
| SKB_NUMPAD | Switches keyboard to partial keyboard of ESC, TAB, SHIFT, and numeric keypad only. |

Only one of SKB_BASIC, SKB_FULL, or SKB_NUMPAD can be used at any one time.

If *lpSKBInfo* is specified for SKB_SHOW or SKB_HIDE, the structure returned contains values that were active before any action requested by the current command.

Any user action on the keyboard itself overrides the function requests. That is, if the user closes the on-screen keyboard, all registered applications become unregistered. If the user minimizes the keyboard, the active **SKBInfo** structure is changed to reflect the new state.

Registration information for 20 window handles is tracked by the function. If one application displays the keyboard and then another one does, both applications must request that the keyboard be hidden before it actually goes away.

Whenever the keyboard display changes, a WM_SKB message is posted to all top-level windows on the desktop. This occurs, for example, upon a change affecting the position or visibility of the keyboard or keypad display, or when the display is restored or minimized. The *wParam* value is SKN_CHANGED, and the LOWORD of *lParam* is a combination of one or more of the following: SKN_POSCHANGED, SKN_VISCHANGED, SKN_PADCHANGED, or SKN_MINCHANGED. The HIWORD value is the window handle of the keyboard.

## Button Bitmaps

The following three bitmaps are provided for owner-drawn pushbuttons that can be used to invoke the on-screen keyboard. The application will process WM_DRAWITEM and other button-related code. On-screen keyboard pushbuttons should behave as other standard buttons (for example, the Minimize button) and invoke their action on button up.

```
#define OBM_SKBBTNUP          32767
#define OBM_SKBBTNDOWN        32766
#define OBM_SKBBTNDISABLED    32765
```

The up bitmap, for example, can be loaded as follows:

```
HANDLE hDLL = GetSystemMetrics(SM_PENWINDOWS);
HBITMAP hBitmap = LoadBitmap(hDLL,
    MAKEINTRESOURCE(OBM_SKBBTNUP));
```

The application must call the **DeleteObject** function to delete each bitmap handle returned by the **LoadBitmap** function.

The button should be left in the up state after it is released; that way, if the user closes the keyboard, the button will be up the next time the keyboard is opened.

The following code segment can be used to get the current keyboard and restore the current state:

```
#include <penwin.h>
    {
    if (ShowKeyboard(hwnd, SKB_SHOW, NULL, NULL))  // nonzero: no error
      {
      // do some actions
      ShowKeyboard(hwnd, SKB_HIDE, NULL, NULL);
      }
    else
      UserErrMsg("Unable to use Screen Keyboard");
    }
```

The following code segment can be used to move the keyboard and then put it back:

```
  {
  SKBINFO skbinfo;
  WORD wCommand = SKB_SHOW | SKB_MOVE;
  POINT pnt;
  pnt.x = wSKBLeft;    // init
  pnt.y = wSKBTop;

  // Show the keyboard
  ShowKeyboard(hwnd, wCommand, &pnt, &skbinfo);
  .
  .   // Other code
  .
```

```
// 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);
}
```

**Return Value** **ShowKeyboard** returns TRUE if successful; otherwise, it returns FALSE.

# SymbolToCharacter

**Action** Converts an array of SYVs to an ANSI string

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL SymbolToCharacter**(*lpsyv*, *cSyv*, *lpstr*, *lpnConv*)

| Parameter | Type | Description |
|---|---|---|
| *lpsyv* | **LPSYV** | Array of symbols. |
| *cSyv* | **int** | Count of symbols. |
| *lpstr* | **LPSTR** | ANSI string buffer. The buffer should be big enough to hold at least *cSyv* number of ANSI characters (including SYV_NULL). |
| *lpnConv* | **LPINT** | If not NULL, *lpnConv* contains the number of symbols converted. If NULL, this parameter is ignored. |

**Comments** This function takes in an array of SYVs and a count of how many SYVs to convert. The buffer *lpstr* should be big enough to hold the total number of converted bytes. For ANSI characters, this value will be *cSyv* bytes. For double-byte characters (Kanji, for example), this value will be (2 * *cSyv)* bytes.

The **SymbolToCharacter** function converts, at most, *cSyv* number of SYVs from *lpsyv* and places them in the *lpstr* buffer. The conversion proceeds until a SYV_NULL value is encountered or until *cSyv* symbols have been converted. A SYV_NULL is converted to `\0`. The actual number of symbols converted is returned in *lpnConv* if *lpnConv* is not NULL.

**Return Value** **SymbolToCharacter** returns FALSE if it encounters one or more symbols that cannot be converted to ANSI. Otherwise, it returns TRUE.

**See Also** **CharacterToSymbol**

# TPtoDP

**Action** Converts an array of points from tablet coordinates to screen coordinates

**Module** RC Manager

**Called By** Application

**Syntax** **BOOL TPtoDP**(*lpPnt*, *cPnt*)

| Parameter | Type | Description |
|---|---|---|
| *lpPnt* | **LPPOINT** | Array of points |
| *cPnt* | **int** | Number of points |

**Comments** The conversion will fail 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 complete inverses of each other.

**Return Value** **TPtoDP** returns FALSE if not all points can be converted.

**See Also** **DPtoTP**

# TrainContext

**Action** Gives the recognizer a previous recognition result that may contain errors plus the correct interpretation of the raw data

**Module** RC Manager

**Called By** System applications (Pen Palette)

**Syntax** **BOOL TrainContext**(*lprcresult, lpyse, csye, lpsyc, csyc*)

| Parameter | Type | Description |
|---|---|---|
| *lprcresult* | **LPRCRESULT** | Pointer to the **RCRESULT** structure containing the **hpendata** that contains the raw data and the recognizer's original interpretation of that data. |
| *lpsye* | **LPSYE** | An array of SYEs that specify the correct interpretation of the raw data. The *lpyse->iSyc* values index the SYCs in the *lpsyc* parameter. |
| *csye* | **int** | The number of SYEs in the *lpsye* array. |
| *lpsyc* | **LPSYC** | An array of SYCs that establish the mapping between the raw data and the characters in the **hpendata** field of the *lprcresult* parameter. |
| *csyc* | **int** | The number of SYCs in the *lpsyc* array. |

**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 is provided by **TrainInk**.

**TrainContext** calls the recognizer-supplied function **TrainContextInternal** in the recognizer DLL identified by *lprcresult->lprc->hrec*. A custom recognizer must provide this function, as well as **TrainInkInternal,** but it can simply return FALSE 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 recognizer, it uses the **TrainContext** function. The trainer incorporated in the Microsoft Pen Palette uses this function for training.

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** field of *lprcresult*. The *lprcresult* parameter must be non-NULL.

In addition to providing the incorrect interpretation of the data (by means of the symbol graph, the **lpsyg** field in *lprcresult*), a more detailed, correct interpretation is also provided

by the SYEs and SYCs. Because the correct interpretation is passed by SYEs, 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 SYCs that point to the ink of the "lc" and, second, the two SYEs with the SYV values "l" and "c." These two SYEs 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 equal to "k" and an array of SYCs that incorporate the ink the recognizer had previously assigned to the "l" and the "c."

To train several SYVs to a single piece of ink (for example, a long stroke that is an "he" ligature), there will be two consecutive SYEs—one for the "h" and one for the "e." Both SYEs will have the same **iSyc** field; this means that these SYEs both point to the same ink. A recognizer will need to take this into consideration to avoid training the two characters separately and using the same ink for both; that would result in having "he" trained as "he he."

## Custom Training

A recognizer may supply its own custom training dialogs. An application should check to see if the recognizer supports custom training by calling **ConfigRecognizer** with the WCR_TRAIN subfunction.

**Return Value**

**TrainContext** returns TRUE if the ink could be trained; otherwise, it returns FALSE.

The trainer does not display an error message if **TrainInk** or **TrainContext** returns FALSE. Error messages that occur when training fails are the responsibility of the recognizer.

**See Also**

**ConfigRecognizer, TrainContextInternal, TrainInk, SYC, SYE** structures

# TrainContextInternal

**Action** At the DLL recognition level, gives the recognizer a previous recognition result that may contain errors plus the correct interpretation of the raw data

**Module** Recognizer

**Called By** RC Manager

**Syntax** **BOOL TrainContextInternal**(*lprcresult, lpsye, csye, lpsyc, csyc*)

| Parameter | Type | Description |
|---|---|---|
| *lprcresult* | **LPRCRESULT** | Pointer to the **RCRESULT** structure containing the **hpendata** that contains the raw data and the recognizer's original interpretation of that data. |
| *lpsye* | **LPSYE** | An array of SYEs that specify the correct interpretation of the raw data. The *lpyse->iSyc* values index the SYCs in the *lpsyc* parameter. |
| *csye* | **int** | The number of SYEs in the *lpsye* array. |
| *lpsyc* | **LPSYC** | An array of SYCs that establish the mapping between the raw data and the characters in the **hpendata** field of the *lprcresult* parameter. |
| *csyc* | **int** | The number of SYCs in the *lpsyc* array. |

**Comments** **TrainContextInternal** is the function in the recognizer DLL that performs the **TrainContext** function. A custom recognizer must provide this function, as well as **TrainInkInternal,** but it may simply return FALSE if the recognizer does not support this type of training.

For details on training using contextual (recognition) information, see the entry for **TrainContext,** earlier in this chapter.

A recognizer that cannot make use of contextual information can instead translate this function into a call to the simpler training function **TrainInkInternal**. The *lprc* argument for **TrainInkInternal** can be taken directly from the *lprcresult.* To create the *hpendata* argument, you first use **CreatePenData** to create a new **HPENDATA** structure; next, step through the SYEs of *lpsye*, using **GetPenDataStroke** to get a copy of the strokes for the characters to be trained from the **hpendata** field of *lprcresult;* finally, use **AddPointsPenData** to insert the pen data into the new **HPENDATA** structure. The *lpsyv* argument for **TrainInk** can be generated from the **syv** fields of the SYEs.

### Saving Training

The recognizer is responsible for saving the results of any training. Two appropriate times to save this information are:

- In response to a WCR_TRAINSAVE subfunction request in **ConfigRecognizer**
- In response to a WCR_RCCHANGE subfunction request in **ConfigRecognizer**

**Return Value** **TrainContextInternal** returns TRUE if the ink could be trained; otherwise, it returns FALSE.

**See Also** **TrainContext, TrainInk, TrainInkInternal, SYC, SYE** structures

# TrainInk

**Action** Gives the recognizer raw data and a correct interpretation of the data

**Module** RC Manager

**Called By** System applications (Pen Palette, Control Panel, and others)

**Syntax** **BOOL TrainInk**(*lprc, hpendata, lpsyv*)

| Parameter | Type | Description |
|---|---|---|
| *lprc* | **LPRC** | Pointer to the **RC** structure. |
| *hpendata* | **HPENDATA** | Pen data to train. This parameter must be non-NULL. |
| *lpsyv* | **LPSYV** | An array of SYVs terminated by SYV_NULL. This parameter must be non-NULL. |

**Comments** **TrainInk** is called by an application with raw data along with a correct interpretation, so that the recognizer can learn to improve subsequent recognition. A second, more complex training function is provided by **TrainContext**.

**TrainInk** is called by an application to access the **TrainInkInternal** function in the recognizer DLL. A custom recognizer must provide this function, as well as **TrainContextInternal,** but it can simply return FALSE if the recognizer does not support this type of training.

**TrainInk** provides the lowest level of basic shape training. It effectively says to the recognizer, "Take the ink in *hpendata* and give it the meaning in *lpsyv*." The recognizer should interpret the ink to meet that request.

In the most common case, *lpsyv* contains a single character, and the recognizer will train a new shape based on the ink and that character. In other cases, multiple SYVs may be passed, indicating that the ink represents multiple characters. The recognizer must decide whether simply to add a new shape with a multiple-SYV meaning or to segment the ink into separate shapes for each SYV.

If the application passes NULL in the *lprc* parameter, the RC Manager substitutes a pointer to the global **RC** structure before passing it on to the recognizer.

The recognizer identified by *lprc->hrec* is called. If *lprc* is NULL or if *lprc->hrec* is RC_WDEFAULT, the recognizer in the global **RC** is called.

An application should check to see if the recognizer supports training by calling **ConfigRecognizer** with the WCR_TRAIN subfunction.

**Return Value** **TrainInk** returns TRUE if the ink described by *hpendata* could be trained; otherwise, it returns FALSE.

The trainer will not display an error message if **TrainInk** or **TrainContext** returns

FALSE. Error messages that occur when training fails are the responsibility of the recognizer.

**See Also** **ConfigRecognizer, TrainContext, TrainContextInternal, TrainInkInternal**

# TrainInkInternal

**Action** At the DLL recognition level, informs the recognizer that the raw data input represents the symbol value results

**Module** Recognizer

**Called By** RC Manager

**Syntax** **BOOL TrainInkInternal**(*lprc, hpendata, lpsyv*)

| Parameter | Type | Description |
|---|---|---|
| *lprc* | **LPRC** | Pointer to the **RC** structure. This parameter must be non-NULL. |
| *hpendata* | **HPENDATA** | Pen data to train. |
| *lpsyv* | **LPSYV** | An array of SYVs terminated by SYV_NULL. |

**Comments** **TrainInkInternal** is the function in the recognizer DLL that is called by the **TrainInk** function with raw data along with a correct interpretation so that the recognizer can learn to improve subsequent recognition. A custom recognizer must provide this function, as well as **TrainContextInternal,** but it can simply return FALSE if the recognizer does not support this type of training.

## Saving Training

The recognizer is responsible for saving the results of any training. Two appropriate times to save this information are:

- In response to a WCR_TRAINSAVE subfunction request in **ConfigRecognizer**
- In response to a WCR_RCCHANGE subfunction request in **ConfigRecognizer**

See the description of **TrainInk** for details on using this function for context-free training.

**Return Value** **TrainInkInternal** returns TRUE if the ink described by *hpendata* could be trained; otherwise, it returns FALSE.

**See Also** **TrainContext, TrainContextInternal, TrainInk**

# UninstallRecognizer

**Action** Unloads the specified recognizer

**Module** RC Manager

**Called By** Application

**Syntax** **void UninstallRecognizer**(*hrec*)

| Parameter | Type | Description |
|---|---|---|
| *hrec* | **HREC** | Recognizer handle |

**Comments** Windows maintains a use count so that the library is not actually unloaded until all callers of **InstallRecognizer** in the given library have called this function. This function should be called by any application one time for every call made to **InstallRecognizer**.

Before attempting to unload the library, **CloseRecognizer** is called within the recognizer.

It is not necessary to uninstall the default recognizer. An application must uninstall all recognizers that it explicitly loads, however.

**Return Value** None

**See Also** **InstallRecognizer**

# UpdatePenInfo

**Action** Called by the pen driver any time a **PENINFO** value changes

**Module** RC Manager

**Called By** Pen driver

**Syntax** **void UpdatePenInfo**(*lppeninfo*)

| Parameter | Type | Description |
| --- | --- | --- |
| *lppeninfo* | **LPPENINFO** | Pointer to the **PENINFO** structure |

**Comments** A **PENINFO** value could change if the driver parameters are altered by the configuration dialog. When this happens, the pen driver must call **UpdatePenInfo** to notify the RC Manager of the change.

**Return Value** None

# Pen Structures

This chapter describes all of the structures that you use in conjunction with the Pen API functions. The entries are listed alphabetically; each includes the code from its header file and a complete description of the structure.

Chapter 9, "Pen API Reference," describes all the Pen API functions.

Chapter 11, "Pen Messages and Constants," describes the messages and constants used in pen computing.

# BOXLAYOUT

Use the **BOXLAYOUT** structure to specify some of the characteristics of a bedit control. The **GUIDE** structure specifies the rest. The HE_GETLAYOUT and HE_SETLAYOUT *wParam* values of the WM_HEDITCTL message can be used to get and set the **BOXLAYOUT** structure for a bedit control.

For more details, see the entry for WM_HEDITCTL messages in Chapter 11, "Pen Messages and Constants."

```
typedef struct
{
int cyCusp;    // Height of the box in pixels when BXS_RECT is specified;
               // height of the cusp in pixels otherwise (in comb style).
int cyEndCusp;       // Height of cusps at either extreme in pixels.
UINT style;          // Style of box edit control;
                       // a combination of the BXS_ flags.
DWORD rgbText;       // Color of the text, -1 = color of window text.
DWORD rgbBox;        // Color of boxes, -1 = color of window frame.
DWORD rgbSelect;     // Color of selection, -1 = color of window text.
} BOXLAYOUT, FAR *LPBOXLAYOUT;
```

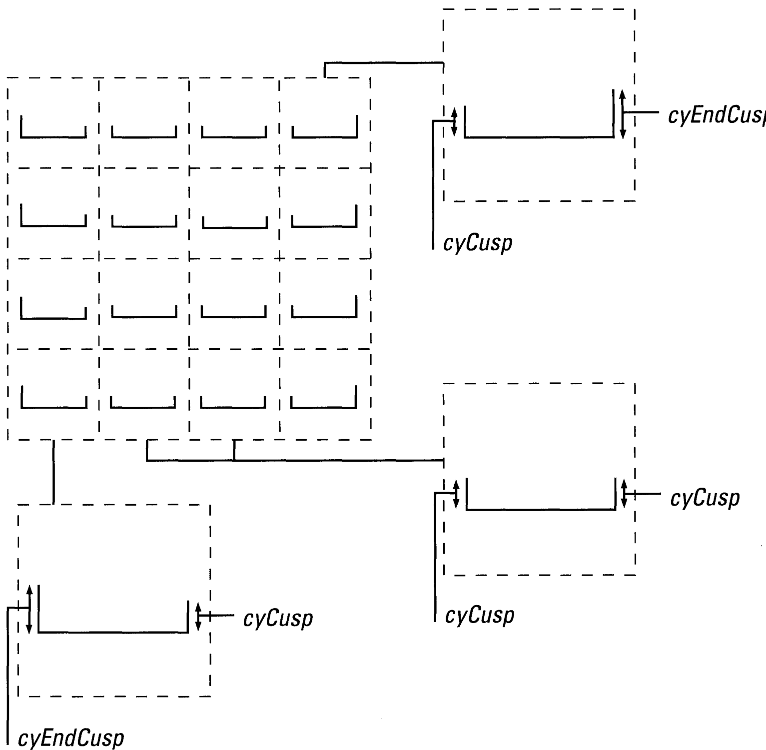
The following table lists the default values for the **BOXLAYOUT** structure.

| Field | Default value |
|---|---|
| **cyCusp** | Equivalent in pixels of BXD_CUSPHEIGHT dialog units |
| **cyEndCusp** | Equivalent in pixels of BXD_ENDCUSPHEIGHT dialog units |
| **style** | 0 for a single-line boxed edit control; BXS_ENDTEXTMARK for a multiline boxed edit control |
| **rgbText** | –1 (Color of window text is used) |
| **rgbBox** | –1 (Color of window frame is used) |
| **rgbSelect** | –1 (Color of window text is used) |

The following table lists the values for the BXS_ style flags.

| BXS_ Flags for Style Field | Meaning |
|---|---|
| BXS_RECT | If this flag is set, rectangular boxes are used; otherwise, combs are used. |
| BXS_ENDTEXTMARK | If this flag is set, an end-of-text marker is displayed in the control. |

Figure 10.1 shows the general layout of a boxed edit control. Some of the terms in this figure are explained under the **GUIDE** structure later in this chapter. To see an individual cell from a boxed edit control, see Figure 10.3.



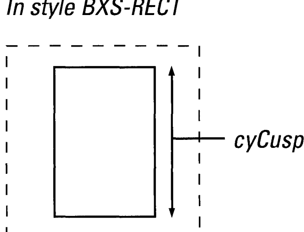




Figure 10.1. Boxed edit control.

# GUIDE

Use the **GUIDE** structure to specify the characteristics of any guidelines used in the writing area.

```
typedef struct
  {
  int xOrigin;        // Position of left edge of first box
                        // in screen coordinates.
  int yOrigin;        // Position of top edge of first box
                        // in screen coordinates.
  int cxBox;          // Width of boxes.
  int cyBox;          // Height of boxes.
  int cxBase;         // Offset of the visible edge of the baseline
                      // within the box from the edge of the box.
  int cyBase;         // Offset of the baseline for writing
                      // from the top of the box
  int cHorzBox;       // Count of columns of boxes.
  int cVertBox;       // Count of rows of boxes.
  int cyMid;          // Offset from baseline to midline.
                        // Zero if not present.
  }
  GUIDE, FAR * LPGUIDE;               // Guidelines for recognizer.
```

The **GUIDE** structure is a part of the **RC** structure, described later in this chapter.

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 look of a boxed edit control is determined by the **BOXLAYOUT** and **GUIDE** structures together.

The **xOrigin** and **yOrigin** fields are screen coordinates of the top-left corner of the area to write in. The **cyBox** and **cxBox** fields are the height and width of the individual boxes to write in. The **cHorzBox** and **cVertBox** fields are the number of columns and rows. A baseline within the box can be indicated by setting **cyBase**. Setting **cyBase** to zero indicates that no baseline is given. The **cxBase** field can indicate a horizontal displacement of the edge of the guideline from the edge of the box where writing can be expected to start.

If only horizontal lines are present, **cxBox** will be zero. In this case, only **yOrigin**, **cyBox**, **cyBase**, and **cyMid** will be valid. If both vertical and horizontal lines are present (boxed input), the RCO_BOXED flag must be set in the **lRcOptions** field of the **RC** structure.

Writing need not be restricted to inside boxes. Inking is still restricted to **rc.rectBound** only. If there is a midline (shown in the bottom row of boxes in Figure 10.2), its height above the baseline is given by **cyMid**. This is zero if no midline exists.

In the **RC** structure, these values will be converted to tablet coordinates before they are passed on to the recognizer. With an orientation of RCOR_LEFT or RCOR_RIGHT, the roles of **cHorzBox** and **cVertBox** are reversed.

The **InitRC** function sets all elements of the **GUIDE** structure to 0.

For boxed input, the result message contains an index to the box containing the first input character. This is numbered in row-major order, zero-based. In Figure 10.2, for example, "h" is in box 12.

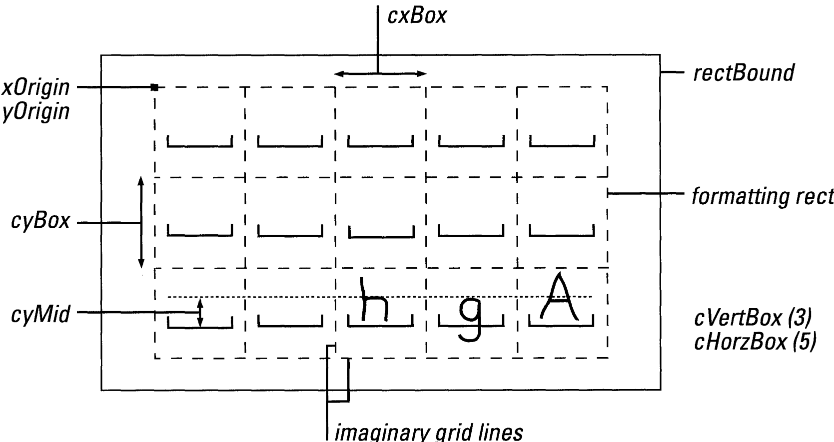



Figure 10.2. Guidelines.

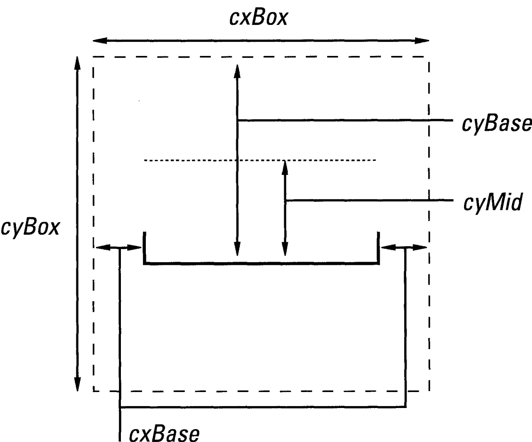



Figure 10.3. Guidelines box.

To change a standard bedit, first make an HE_GETRC subfunction request of WM_HEDITCTRL to get the **GUIDE** structure. Make any changes needed, and then use HE_SETRC to inform the bedit control of the changes.

For best recognition results, the pair-wise ratios of **cxBox**, **cyBox**, and **cyBase** should be similar to the default ratios.

# OEMPENINFO

```
typedef struct
{
UINT wPdt;            // Pen data type.
UINT wValueMax;       // Largest value returnable by device.
UINT wDistinct;       // Number of distinct readings possible.
} OEMPENINFO;
```

The **OEMPENINFO** structure contains a description of the additional OEM information that the hardware can generate. It is a part of the **PENINFO** structure described later in this chapter.

Besides capturing the (x,y) data, a pen device has the option of supporting a number of other types of input data: pressure, height, angle, and so on. A pen driver can capture up to six other types of data. An application can access this data through the **GetPenHwData** function.

Each event from the pen generates a position as well as information on any of the other types of data. The width of this optional data is **cbOemData** 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 **OEMPENINFO** field in the **PENINFO** structure. PDT_NULL indicates no data. Values above PDT_OEMSPECIFIC are reserved for private use by drivers for data types not currently defined as standard. The *wValueMax* parameter is the largest value that can be returned by the device for that data type. The *wResolution* parameter is the number of distinct readings the device can take between 0 and *wMaxValue*.

The constants that specify OEM-specific data are listed in the following table.

| Constant | Value |
|---|---|
| PDT_NULL | 0 |
| PDT_PRESSURE | 1 |
| PDT_HEIGHT | 2 |
| PDT_ANGLEXY | 3 |
| PDT_ANGLEZ | 4 |
| PDT_BARRELROTATION | 5 |
| PDT_OEMSPECIFIC | 16 |

The units for height are 0.01 cm.

The units for the angle measure are in tenths of a degree. Zero for ANGLEZ indicates that the pen is perpendicular to the writing surface. Zero for ANGLEXY indicates that the pen is parallel with the side, running from the top to the bottom of the writing surface. The zero position for barrel rotation is device-dependent.

Figure 10.4 shows the pen in a position where both angles are roughly 45 degrees.

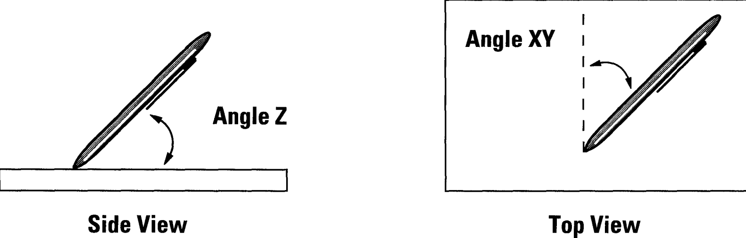



Figure 10.4. Pen angles relative to the surface of the tablet.

As an example, consider a device that can sense height and ANGLEZ. It 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 **rgOemPenInfo** for this device would 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,y) data. There must be a one-to-one correspondence between the OEM event data and the (x,y) data.

See the **PENINFO** structure for additional information.

# PENDATAHEADER

```
typedef struct
  {
  UINT wVersion;          // Pen data format version
  UINT cbSizeUsed;        // Size of pen data memory block
  UINT cStrokes;          // Number of strokes (each pen up and
                              // pen down run is a single stroke)
  UINT cPnt;              // Total count of points
  UINT cPntStrokeMax;     // Length of longest stroke in points
  RECT rectBound;         // Coordinates of bounding rectangle of down points
  UINT wPndts;            // State of the various PDTS_ bits
  int nInkWidth;            // Ink width
  DWORD rgbInk;         // Ink color
  }
  PENDATAHEADER, FAR * LPPENDATAHEADER;
```

Use the **PENDATAHEADER** structure in conjunction with the **GetPenDataHeader** function. The structure describes the specified pen data memory block.

If data is collected outside the bounding rectangle, the **rectBound** field reflects this. This means that **rcresult.rectBoundInk** is outside the rectangle and does not truly reflect the bounds of the ink on the screen. However, it does reflect the size of the object if it is to be drawn in a different window.

The following table lists the PDTS_ values for the **wPndts** field.

| PDTS_ bits for wPndt | Meaning |
|---|---|
| PDTS_ARBITRARY | The application has done its own scaling of the data point. |
| PDTS_COMPRESS2NDDERIV | The second derivative between points is stored. |
| PDTS_COMPRESSED | The data is compressed. |
| PDTS_COMPRESSMETHOD | Bits have been saved to encode which compression scheme is used. |
| 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 mm. Positive x is to the right; positive y is down. |

| **PDTS_ bits for wPndt** | **Meaning** |
|---|---|
| PDTS_LOMETRIC | Each logical unit is mapped to 0.01 mm. Positive x is to the right; positive y is down. |
| PDTS_NOOEMDATA | No OEM data is present. |
| PDTS_NOPENINFO | The **PENINFO** structure has been trimmed from the header. |
| PDTS_NOUPPOINTS | The data points on pen up have been removed. |
| PDTS_SCALEMASK | This value refers to the bits used to mask scaling. |
| PDTS_SCALEMAX | This value represents the maximum scaling allowed. |
| PDTS_STANDARDSCALE | The standard scaling metric is equivalent to PDTS_HIENGLISH. Standard recognizers will scale to this. |

# PENINFO

```
#define MAXOEMDATAWORDS  6

typedef struct
  {
  UINT cxRawWidth; // Max X coordinate & width of tablet in 0.001 inches
  UINT cyRawHeight;  // Max Y coordinate & height of tablet in 0.001 inches

  UINT wDistinctWidth; // Number of distinct X values tablet returns
  UINT wDistinctHeight;  // Number of distinct Y values tablet returns

  int nSamplingRate;   // Samples/second
  int nSamplingDist;   // Minimum distance moved in either direction
            // to generate a new pen event

  LONG lPdc;      // Pen device capabilities
  int cPens;      // Number of pens supported

  int cbOemData;   // Width of OEM data packet
  OEMPENINFO rgoempeninfo[MAXOEMDATAWORDS];  // Supported OEM data types

  UINT rgwReserved[8];   // For internal use
  }
  PENINFO, FAR *LPPENINFO;
```

The DRV_GetPenInfo pen driver message fills the **PENINFO** structure with the current pen parameters. DRV_GetPenInfo returns FALSE if a tablet is not physically present. If this occurs, the **peninfo** field may not be present.

The following table lists the PDC_ values for the **lPdc** field.

| Value | Meaning |
|---|---|
| PDC_BARREL1 | Barrel button 1 is present. |
| PDC_BARREL2 | Barrel button 2 is present. |
| PDC_BARREL3 | Barrel button 3 is present. |
| PDC_INTEGRATED | The display and the digitizer share the same surface. |
| PDC_INVERT | The pen can detect that the other end of the pen is in contact with the tablet. |
| PDC_PROXIMITY | The pen position can be detected without the tip's being in contact with the tablet. |
| PDC_RANGE | The pen can generate an event upon leaving or entering the detection range. |
| PDC_RELATIVE | The physical device is a relative motion device only. |

## The PENINFO Fields

The following paragraphs discuss the **PENINFO** fields, listed in the order in which they appear in the preceding structure.

### cxRawWidth, cxRawHeight

```
UINT cxRawWidth;
UINT cyRawHeight;
```

The **cxRawWidth** and **cyRawHeight** fields describe the physical tablet dimensions in thousandths of an inch. These values also specify the maximum x and y table coordinates.

### wDistinctWidth, wDistinctHeight

```
UINT wDistinctWidth;
UINT wDistinctHeight;
```

The **wDistinctWidth** and **wDistinctHeight** fields specify the number of distinct values that the hardware can detect. For example, if a tablet is 8 inches wide and has a resolution of 1/500 of an inch, **cxRowWidth** is 8000 and **wDistinctWidth** is 4000, because the tablet hardware can return 4000 distinct x values ranging from 0 to 8000.

### nSamplingRate

```
int nSamplingRate;
```

The **nSamplingRate** field specifies the number of times per second that the pen hardware is sampled. This rate can be adjusted with the DRV_SetSamplingRate pen driver message.

### nSamplingDist

```
int nSamplingDist;
```

The **nSamplingDist** field specifies the distance in distinct tablet units a pen must travel before a new pen event is generated. This distance can be adjusted with the DRV_SetSamplingDistance pen driver message.

### lPdc

```
LONG lPdc;
```

The **lPdc** field is the combination (OR operator) of all of the PDC_ (pen device capabilities) bits that describe the tablet capabilities.

### cPens

```
int cPens;
```

The **cPens** field specifies the number of pens the tablet can support.

## cbOemData

```
int cbOemData;
```

The **cbOemData** field specifies the width, in bytes, of the additional OEM data that is passed in each pen packet. For example, if a tablet supports pressure and angle Z information, this is two additional words of OEM information, so **cbOemData** is 4.

## rgoempeninfo

```
OEMPENINFO rgoempeninfo[MAXOEMDATAWORDS];
```

The **rgoempeninfo** field is is an array of **OEMPENINFO** structures. Each structure describes one word of additional OEM data contained in each pen packet. For additional information, see the description of **OEMPENINFO**, earlier in this chapter.

## rgwReserved

```
UINT rgwReserved[8];
```

This array is reserved for future use.

# PENPACKET

```
#define MAXOEMDATAWORDS   6

typedef struct
  {
  UINT wTabletX;                        // X in raw coordinates.
  UINT wTabletY;                        // Y in raw coordinates.
  UINT wPdk;                            // State bits.
  UINT rgwOemData[MAXOEMDATAWORDS];     // OEM-specific data.
  }
  PENPACKET, FAR * LPPENPACKET;
```

The basic unit of communication between the pen driver and Windows is a *pen packet.* 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. Many physical events—interrupts—may be needed to construct a single logical event.

The **PENPACKET** structure is formally defined in the TABLET.INC file. The following code fragment contains the definition:

```
; PENPACKET - what drivers should use to communicate with
; PenWin.Dll
;

PENPACKET struc
  wTabletX     dw    0;    X in tablet coordinates
  wTabletY     dw    0;    Y in tablet coordinates
  wPDK         dw    0;    various status bits for packet
  rgwOemData   dw    MAXOEMDATAWORDS dup (0);OEM info like pressure
PENPACKET ends
```

The **rgwOemData** field contains the real-time values associated with the pen data types described in the **OEMPENINFO** structure.

# RC

The core of the recognition process is the **RC** (Recognition Context) structure. Although there are a large number of parameters in the **RC** structure, an application will only have to deal with a few of them. The application uses the **InitRC** function to set the default values and then adjusts certain parameters before making one of the **Recognize** calls.

```
typedef struct
  {
  HREC hrec;               // Handle of recognizer to use.
  HWND hwnd;               // Window to send results to.
  UINT wEventRef;          // Data reference indicating first
                             // point to use in recognition.
  UINT wRcPreferences;     // Preferences.
  LONG lRcOptions;         // Options.
  RCYIELDPROC lpfnYield; // Procedure called during yield.

  BYTE lpUser[cbRcUserMax];        // Current writer.
  UINT wCountry;                 // Country code.
  UINT wIntlPreferences;         // RCIP_ Flags
  char lpLanguage[cbRcLanguageMax]; // language strings.

  LPDF rglpdf[MAXDICTIONARIES];      // List of dictionary functions.
  UINT wTryDictionary;         // Maximum enumerations to search.
  CL clErrorLevel;   // Level at which recognizer should reject input.

  ALC alc;           // Enabled alphabet.
  ALC alcPriority;   // Prioritizes the ALC_ codes used to enable alphabets.

  BYTE rgbfAlc[cbRcrgbfAlcMax];// Bit field for enabled characters.
  UINT wResultMode;  // When to send (ASAP or when recognition complete).

  // Control of recognition completion.
  UINT wTimeOut;     // Time-out threshold in milliseconds
                         // (after this time, stop recognition).
  LONG lPcm;         // Flags for ending recognition.

  RECT rectBound;    // Bounding rectangle for inking.
  RECT rectExclude;  // Pen down inside this ends recognition.

  GUIDE guide;       // Define guidelines for recognizer.

  UINT wRcOrient;    // Orientation of writing relative to
                       // tablet coordinates.
  UINT wRcDirect;    // Direction of writing.
  int nInkWidth;     // Ink width.  0 (no ink) to 15.
  COLORREF rgbInk;     // Color of ink.
```

```
DWORD dwAppParam;       // For use by application.
DWORD dwDictParam;      // For use by application to be
                          // passed on to dictionaries.
DWORD dwRecognizer;     // For use by an application to pass
                          // information to recognizer.

UINT rgwReserved[cwRcReservedMax];  // Reserved for future
                                          //  use by Windows.
}
RC, FAR * LPRC;         // Recognition context.
```

## The RC Fields

The following paragraphs discuss the **RC** fields, listed in the order in which they appear in the preceding structure.

### rc.hrec

```
HREC hrec;
```

The **hrec** field is the handle of the recognizer to use. This value should be set to the value returned by a previous call to **InstallRecognizer** or RC_WDEFAULT for the default recognizer.

If **hrec** is NULL, the null recognizer is used. WM_RCRESULT messages are generated—as with a real recognizer—but the **wResultsType** field of **RCRESULT** is set to RCRT_NORECOG, and the **hSyv** and **lpSyv** fields are set to NULL.

### rc.hwnd

```
HWND hwnd;
```

The **hwnd** field specifies the window to send recognition results to. This field cannot be NULL. Also, the mouse capture will be set to this window to clear the queue of pending mouse messages that were meant for recognition.

### rc.wEventRef

```
UINT wEventRef;
```

The value for **wEventRef** indicates which tablet data to begin recognition with. For a fuller explanation of the **wEventRef** field, see the entry for the **GetMessageExtraInfo** function in Chapter 9, "Pen API Reference."

**InitRC** sets this field to RC_WDEFAULT. If **Recognize** is called during the processing of the mouse message (such as WM_LBUTTONDOWN), triggering recognition, the application need take no other action.

Before an application starts recognition on some other Windows event, it should save the **wEventRef** from the appropriate mouse message (using **GetMessageExtraInfo**) and place this value in **wEventRef** before calling **Recognize**.

This field is not used on calls to **RecognizeData**.

## rc.wRcPreferences

```
UINT wRcPreferences;
```

The **wRcPreferences** field specifies the user preferences as a combination of the RCP_ constants described in the following table.

| Value | Meaning |
|---|---|
| RCP_LEFTHAND | User writes with left hand. The default assumption is right-handed. |
| RCP_MAPCHAR | This value tells the recognizer to fill in segmentation information in the **lpsyc** field. This cannot be set by the user, because there is no Control Panel access to this value. It is set if training is enabled. |

## rc.lRcOptions

```
LONG lRcOptions;
```

The **lRcOptions** field specifies various options for recognition. It is a logical OR combination of any of the RCO_ constants described in Chapter 11, "Pen Messages and Constants."

## rc.lpfnYield

```
RCYIELDPROC lpfnYield;
```

The **lpfnYield** field is 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, and 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 **rc.lpfnYield** set to RC_LDEFAULT, then the default yield function is called. If the **rc.lpfnYield** field is not NULL, the recognizer will call **lpfnYield** every time before it yields.

## rc.lpUser

```
#define cbRcUserMax       32
BYTE lpUser[cbRcUserMax];
```

The **lpUser** field 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** field is NULL, it means that the recognizer will use the standard prototype set—the prototype set as it existed before anyone modified it (through training, for example).

## rc.wCountry

```
UINT wCountry;
```

The **wCountry** field 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** field in the [intl] section of the WIN.INI file.

## rc.wIntlPreferences

```
UINT wIntlPreferences;
```

The **wIntlPreferences** field contains a combination of various RCIP_ flags.

Currently, the only RCIP_ flag is RCIP_ALLANSICHAR. This flag specifes that the user intends to use the entire ANSI character set. A recognizer should examine this flag to decide which characters to enable for recognition.

If nothing is set, then only characters from the current language or languages are enabled.

## rc.lpLanguage

```
#define cbRcLanguageMax      44;
char lpLanguage[cbRcLanguageMax];
```

The **lpLanguage** field is a list of language strings. Each string is zero-terminated, and the list ends in the 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** field in the [intl] section of the WIN.INI file. These three-letter codes are documented in the Microsoft Windows SDK.

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** field 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 will still be possible to enter ANSI characters outside the given language through the use of the on-screen keyboard and ALT+numeric keypad combinations.

## rc.rglpdf

```
#define MAXDICTIONARIES 16
LPDF rglpdf[MAXDICTIONARIES]
```

The dictionary path field, **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 **DictionarySearch** in Chapter 9, "Pen API Reference."

## rc.wTryDictionary

```
UINT wTryDictionary;
```

The **wTryDictionary** field 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.

## rc.clErrorLevel

```
CL 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 reprompt the user 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** field allows the application to signal its preference to the recognizer. Recognizers should report the "unknown" symbol for any symbol having a confidence level below **clErrorLevel**.

## rc.alc

```
typedef LONG ALC;
ALC alc;
```

The **alc** field is used to define the enabled alphabet for any **RC** structure. You define the enabled alphabet using the ALC_ constants described in Chapter 11, "Pen Messages and Constants." Any of the ALC_ constants can be combined together to form the set of characters you want.

The actual characters enabled are language-dependent. For example, if the user has requested French language support, "è" would be included in the lowercase alphabet. Likewise, "£" is included in place of "$" in British systems if ALC_MONETARY is set.

Setting the RCIP_ALLANSICHAR flag in the **wIntlPreferences** field 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 field. If you want an application to pass character subset information to private non-ANSI recognizers, you can use the **dwRecognizer** field.

A recognizer will 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 "unknown" if a suitable match is not found.

## rc.alcPriority

```
ALC alcPriority
```

The **alcPriority** field prioritizes the ALC_ codes used to enable alphabets. It does this by telling the recognizer in which order to list options in the symbol graph—alphanumeric or numeric.

The **alcPriority** field uses the same ALC_ codes used in the **alc** field. The bits set in **alcPriority** should be a subset of those set in **alc**. Bit sets in the **alcPriority** that are not also set in the **alc** field have no effect.

A recognizer can recognize a glyph that belongs to more than one enabled ALC_ subset. For example, the "l" glyph can be the letter "l" in the ALC_LCALPHA subset or the number "1" in the ALC_NUMERIC subset. The **alcPriority** field specifies that the recognizer should return those interpretations that are in the subsets indicated in **alcPriority** first. If no interpretations are in any of the **alcPriority** sets, or no priority fields are set, the recognizer will return all possibilites within the enabled sets.

For example, suppose the user writes a symbol that looks like either a "q" or a "9." The symbol graph generated contains `{q | 9 }`. The **alcPriority** field determines the exact look of the symbol graph.

If **alcPriority** = ALC_ALPHA, the recognizer should return `{ q | 9 }` in the symbol graph. If **alcPriority** = ALC_NUMERIC, 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** field indicates which characters have priority.

## rc.rgbfAlc

```
#define cbRcrgbfAlcMax   32
BYTE rgbfAlc[cbRcrgbfAlcMax];
```

The **rgbfAlc** field is the bitfield used for enabled characters. For more details, see the description of enabled alphabets in Chapter 11, "Pen Messages and Constants."

If ALC_USEBITMAP is set, the 256-bit bitfield in **rc.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, the fifth bit of byte four is set:

```
rgbfAlc[4] |= 0x10
```

A recognizer that recognizes characters other than ANSI can ignore this field. If an application wants to pass character subset information to private non-ANSI recognizers, it can use the **dwRecognizer** field of the **RC** structure.

A set of macros simplifies user setting and testing the **rgbfAlc** bits for the **RC** specified by *lprc*. The ANSI macros listed in the following table set (bit=1), reset (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 | Usage |
|---|---|
| **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 **rc.rgbfAlc[]** also requires combining ALC_USEBITMAP by an OR operator with **rc.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 **rc.rgbfAlc** that correspond to 0 through 9. Thus, to recognize octal numbers (the set 0 to 7), the following code can be used:

```
RC rc;
int i;

rc.alc = ALC_USEBITMAP;      /* note no ALC_NUMERIC */
for (i = (int)'0'; i <= (int)'7'; i++)
  SetAlcBitAnsi(&rc, i);
```

## rc.wResultMode

```
UINT wResultMode;
```

The **wResultMode** field specifies the timing and granularity of the results messages to be sent back to the specified window. The following times are defined.

| Value | Meaning |
|---|---|
| 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. |

| Value | Meaning |
|---|---|
| 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. In the Microsoft recognizer, this is implemented only for boxed input. Default dictionary processing is disabled when this value is used. |

Future versions may implement other results modes.

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 "word" returned may contain spaces. This is useful if a space was resolved only by dictionary look-up. 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 how the word will be followed. If the word is followed by a word on essentially the same line, the word should be space-terminated. If the word is followed by text on a new line, it 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.

## rc.wTimeOut

```
UINT wTimeOut;
```

The **wTimeOut** field 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 **rc.lPcm** does not enable time-out.

In general, applications should use the value set by the user with the Control Panel. This value can be set by setting this field to RC_WDEFAULT.

The maximum value allowed is 65,534 milliseconds. If **wTimeOut** is set to FFFF (65,535), the system level value is used.

## rc.lPcm

## rc.rectBound

## rc.rectExclude

```
LONG lPcm;
RECT rectBound;
RECT rectExclude;
```

These three fields of the **RC** structure set the conditions for ending recognition.

The **lPcm** field sets the flags for ending recognition. The two **rect** fields specify inclusive and exclusive rectangles for inking. The rectangle values are in screen coordinates. The rectangle values can be in tablet coordinates if RCO_TABLETCOORD is set. 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 **rc.lPcm**, the first pen down outside the rectangle will close the context. Dragging the pen outside the rectangle after starting inside will not close the context; the data is still collected outside the rectangle.

If PCM_RECTEXCLUDE is set in **rc.lPcm**, any pen down event within **rectExclude** will close the context.

The event that ends pen collection mode—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 sized, the application should respecify the **rectBound** field in the **RC** structure.

The following table lists the values for **lPcm**.

| Value | Meaning |
|---|---|
| PCM_ADDDEFAULTS | If this bit is set, the default ending conditions set in the Global RC are connected by OR operators to the values set by the application. This bit is set by **InitRC** so that Control Panel settings for time-out and proximity will be considered during the subsequent call to **Recognize**. |
| PCM_INVERT | Ends recognition on pen inversion (a pen tap with the opposite end—that is, the blunt end of the pen). |
| PCM_PENUP | Ends pen collection mode if pen is not in contact with the screen (that is, the pen tip is no longer pressed). |
| PCM_RANGE | Ends pen collection mode if the pen leaves the detection range. |
| PCM_RECTBOUND | Ends on pen down outside the bounding rectangle. |
| PCM_RECTEXCLUDE | Ends on pen down inside the exclusion rectangle. |
| PCM_TIMEOUT | Ends on time-out. |

## rc.guide

```
GUIDE guide;
```

The **guide** field is a structure of the **GUIDE** type described earlier in this chapter. It contains information that specifies the placement of guidelines in the writing area for the recognizer's use.

## rc.wRcOrient

```
UINT wRcOrient;
```

The **wRcOrient** field specifies the orientation of the tablet. The RCOR_ constants are used to establish the tablet orientation.

For details about RCOR_ constants, see Chapter 11, "Pen Messages and Constants."

## rc.wRcDirect

```
UINT wRcDirect;
```

The **wRcDirect** field informs the recognizer of the direction of writing. 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 top down (primary) and then right to left across the page (secondary).

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 will respond to this field.

The RCD_ constants used to set this field are described in Chapter 11, "Pen Messages and Constants."

## rc.nInkWidth, rc.rgbInk

```
int nInkWidth;
COLORREF rgbInk;
```

These two fields specify the ink width and color to be used during inking.

The **nInkWidth** field is the thickness, in pixels, of the pen to use during inking. If this value is 0, no ink will be drawn. The current maximum value allowed is 15. The default is the ink width set in the global **RC**.

The **rgbInk** field is the color to use for inking. If this is not a solid color, it will be mapped to the closest solid color. The default is the ink color set in the global **RC**.

## rc.dwAppParam, rc.dwRecognizer

```
DWORD dwAppParam;
DWORD dwRecognizer;
```

These two fields are analogous to the **dwDictParam** field. (See the following entry).

The **dwAppParam** value is provided for use by the application and passed to the application by way of the **lprc** field 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 zero by **InitRC** and should remain zero if they are not used by the application or recognizer.

## rc.dwDictParam

```
DWORD dwDictParam;
```

This 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. For example, you can use this to highlight misspelled words.

If it is not used by the application, **dwDictParam** should be left to the value (0) set by **InitRC**.

## rc.rgwReserved[cwRcReservedMax]

```
UINT rgwReserved[cwRcReservedMax];
```

The **rgwReserved** field, used internally, is reserved for future use. Applications should not change the values set by **InitRC** for this field.

# RCRESULT

A new message, WM_RCRESULT, has been added to Windows to support handwriting. This message is sent to the specified window when the recognizer has a result to return.

The *wParam* parameter of the message is the reason recognition ended (one of the REC_ codes). It is REC_OK if more results are sent; otherwise, it is the same value for the last message as that 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.

```
typedef struct
  {
  SYG syg;              // Symbol graph.

  UINT wResultsType;

  // The next three values are filled in by the RC Manager
  // and the recognizer.
  int cSyv;             // Count of symbols. This is 0 if not able to
                          // recognize results or no recognition is requested.
  LPSYV lpsyv;          // Recognized symbols.  NULL-terminated.
                          // Count in cSyv does not include NULL.
  HANDLE hSyv;          // Global shared handle out of which
                          // lpSyv was allocated.
  int nBaseLine;        // Baseline of input writing.
                          // 0 if not calculated or unknown.
  int nMidLine;         // Midline of input writing.

  HPENDATA hpendata;    // pen data memory block
  RECT rectBoundInk;      // ink data bounds

  POINT pntEnd;           // point that terminated recognition
  LPRC lprc;            // Recognition context used.
  }
  RCRESULT, FAR * LPRCRESULT;      // Recognition result
```

The following table elaborates on the **RCRESULT** fields. All of the fields are allocated with GMEM_SHARE so they can be passed between processes.

| RCRESULT field | Description |
|---|---|
| **syg** | This field 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.<br><br>The **syg.lpsyc** field is not valid unless RCP_MAPCHAR was set in the RC structure when **Recognize** or **RecognizeData** was called.<br><br>For more details, see **SYE, SYG,** and **SYC,** later in this chapter. |
| **wResultsType** | The values for this field are listed in the table following. The values can be connected by OR operators. They are not mutually exclusive. |
| **lpsyv** | This field contains the symbols that are recognized. An application should use these values to display the text or gestures recognized. The **lpsyv** field is the result of any dictionary search on the SYG or further postprocessing. It is NULL if the NULL recognizer is used. |
| **hpendata** | This field 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,** it will not be displayed. However, **rcresult.rectBoundInk** will be calculated as if the ink were drawn.<br><br>If data is collected outside the bounding rectangle, the **rectBound** field of **PENDATAHEADER** will reflect this. (Note that only down points are reflected in **rectBound.)** This means, however, that a portion of **rectBoundInk** will lie 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.<br><br>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. |

| RCRESULT field | Description |
|---|---|
| **lprc** | This is the **RC** used by recognition. Any default values (RC_WDEFAULT or RC_LDEFAULT) are replaced by the correct default value. |

When the recognizer fills the *wResultType* variable in the **RCRESULT** structure, it should choose the appropriate RCRT_ value. The following list is meant to help clarify the options. The recognizer should never have to set RCRT_ALREADYPROCESSED, RCRT_GESTURETOKEYS, or RCRT_GESTURETRANSLATED.

| Value | Meaning |
|---|---|
| RCRT_DEFAULT | The recognition was successful; the **RCRESULT** structure contains valid information. This typically occurs with text that the recognizer readily recognizes. This value can be connected by OR operators to RCRT_GESTURE, RCRT_PRIVATE, and RCRT_UNIDENTIFIED. |
| RCRT_ALREADYPROCESSED | This flag is set by a hook or the Gesture Manager 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 or the Gesture Manager may set this flag. The Hform sample application demonstrates its use.<br><br>RCRT_ALREADYPROCESSED can be connected by OR with RCRT_GESTURE, RCRT_NORECOG, RCRT_PRIVATE, and RCRT_UNIDENTIFIED. |
| RCRT_GESTURE | The result is a gesture symbol. RCRT_GESTURE is not usually combined by OR with anything. |
| RCRT_GESTURETOKEYS | The Gesture Manager translated the gesture to a set of virtual keys. The keys may represent such things as function or navigation keys. |
| RCRT_GESTURETRANSLATED | The Gesture Manager translated the gesture to an ANSI text string. |
| RCRT_NORECOG | Nothing is recognized; the only data in the RCRESULT structure is the raw data in the pen data format. No recognition was attempted. RCRT_NORECOG is not usually combined by OR with anything. |

| Value | Meaning |
|---|---|
| RCRT_NOSYMBOLMATCH | Nothing is recognized. The ink drawn does not match any enabled symbols. This flag must not appear with any other flags; do not combine it with any other flags by OR. |
| RCRT_PRIVATE | The results have a unique meaning to the recognizer, or they are in a special format. This flag can be connected by OR with RCRT_UNIDENTIFIED. |
| RCRT_UNIDENTIFIED | The result contains unidentified results.The RCRT_UNIDENTIFIED bit was set, indicating that some piece of the user's input was not found in the recognizer's database. The **RCRESULT** structure still contains valid information, but not all of it was recognized. This flag can be connected by OR with RCRT_PRIVATE. |

# RECTOFS

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 Delete 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 the 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_HEDITCL 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.

```
typedef struct
  {
  int dLeft;
  int dTop;
  int dRight;
  int dBottom;
  } RECTOFS;   // Rectangle offsets
RECTOFS FAR * LPRECTOFS;
```

# SKBINFO

The **SKBINFO** structure stores the current on-screen keyboard information. You use it with the **ShowKeyboard** function.

```
typedef struct
  {
    HWND hwnd;          // handle to SKB window
    UINT nPad;          // current pad view
    BOOL fVisible;        // Visible status
    BOOL fMinimized;       // Minimized status
    RECT rect;            // Restored keyboard rectangle
    DWORD dwReserved;  // Reserved for future use
  }
  SKBINFO, FAR *LPSKBINFO;
```

The following table describes the **SKBINFO** structure fields.

| Field | Description |
|---|---|
| **hwnd** | Handle to the on-screen keyboard window |
| **nPad** | Current view of the keypad (full, basic or numeric pad) |
| **fVisible** | If TRUE, the on-screen keyboard is available and visible |
| **fMinimized** | If TRUE, the on-screen keyboard is minimized |
| **rect** | The screen coordinates of the restored keyboard rectangle |

# STROKEINFO

The **STROKEINFO** structure serves two main purposes. First, it is returned by the **GetPenHwData** and **GetPenHwEventData** functions with each piece of new data from the tablet. Second, it is used in the pen data functions (**AddPointsPenData** and **GetPenDataStroke**) as a header for each stroke. In both cases, it contains information about a sequence of data from the tablet.

```
typedef struct
  {
  UINT cPnt;     // Count of points in stroke.
  UINT cbPnts;   // Count of bytes used for stroke.
  UINT wPdk;     // State of stroke.
  DWORD dwTick;  // Time of stroke.
  }
  STROKEINFO, FAR * LPSTROKEINFO;  // Stroke header.
```

The following table describes the **STROKEINFO** structure fields.

| Field | Description |
|---|---|
| **cPnt** | Number of points in the stroke. |
| **cbPnts** | Used internally to contain length of compressed data. Applications should ignore this value. |
| **wPdk** | Contains information about the state of the stroke. The **wPdk** field is one or more of the PDK_ bits described in Chapter 11, "Pen Messages and Constants." |
| **dwTick** | Time in milliseconds since the first point in the pen data was collected from the tablet. If an application is creating its own pen data memory block and the timing of the stroke is not important, this field should be set to zero for all strokes. |

# SYG, SYE, SYC, and SYV

Upon completing recognition, a recognizer returns a symbol graph—an **SYG** structure—as a field of the **RCRESULT** structure. 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 a null-terminated array of symbols.

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**s back to the corresponding raw data. If two or more consecutive **SYE**s map to the same **SYC**, they represent an indivisible unit; for example, the user might teach the system a "th" with the crossbar of the "t" connected to the "h". **SYC**s are used primarily for training.

In general, an application does not use the symbol graph directly. Instead, it should use the *lprcresult->hSyv* field, which contains a symbol string that represents the best interpretation from the symbol graph. For details about symbol values (SYVs), see Chapter 11, "Pen Messages and Constants."

**SYE**s and **SYC**s 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 | Meaning |
|---|---|
| **HPENDATA** | Contains raw data information: strokes, pen up, pen down, points, and so on. |
| **SYC** | A symbol character map. **SYC**s delimit strokes in **HPENDATA**. A single shape can be identified by one or more **SYC**s. Each **SYC** identifies a starting stroke, an ending stroke, a starting point, and an ending point. A flag also indicates whether subsequent **SYC**s in the array contain additional strokes for the shape. (This feature is used for delayed strokes, such as "T" crossings.) |
| **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**s; this array identifies the raw data that makes up the symbol. It is possible for several **SYE**s to use the same **SYC**s. The **SYC**s 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 two training functions, **TrainContext** and **TrainInk,** use this information in training.

## SYE (Symbol Element)

```
typedef struct
  {
  SYV syv;              // Symbol value.
  LONG lRecogVal;       // Reserved for use by recognizer.
  CL cl;                // Confidence level of symbol value.
  int iSyc;
  }
  SYE, FAR * LPSYE;               // Symbol graph element.
```

## SYG (Symbol Graph)

```
#define MAXHOTSPOT 8

typedef struct
  {
  POINT rgpntHotSpots[MAXHOTSPOT];     // Hot spots.
  int cHotSpot;             // Number of valid hot spots in array.
  int nFirstBox;            // If RCO_BOXED set, this contains
                              // the index to the box of the first
                              // character in the results. The index is
                              // in row-major order.
  LONG lRecogVal;           // Reserved for use by recognizer.
  LPSYE lpsye;              // Nodes of symbol graph.
  int cSye;                 // Number of SYEs in symbol graph.
  LPSYC lpsyc;
  int cSyc;
  }
  SYG, FAR * LPSYG;                     // Symbol graph.
```

If a single entity recognized by the recognizer is mapped to a string of several symbol values, the recognizer creates multiple **SYE**s. 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.

**Note**

The **nFirstBox** field has no meaning for gestures. A gesture is applied to the location indicated by its hot spot.

## SYC (Symbol Correspondence)

```
typedef struct
  {
  UINT wStrokeFirst;    // First stroke, inclusive.
  UINT wPntFirst;       // First point in first stroke, inclusive.

  UINT wStrokeLast;     // Last stroke, inclusive.
  UINT wPntLast;        // Last point in last stroke, inclusive. Set
                          // to wPntAll for all points.

  BOOL fLastSyc;          //TRUE: no more SYCs follow for current SYE.
  }
  SYC, FAR *LPSYC;
```

Figure 10.5 illustrates the relationship of symbol values and symbol graphs. The first line shows that a symbol value is a single symbol element (an **SYE**). 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.

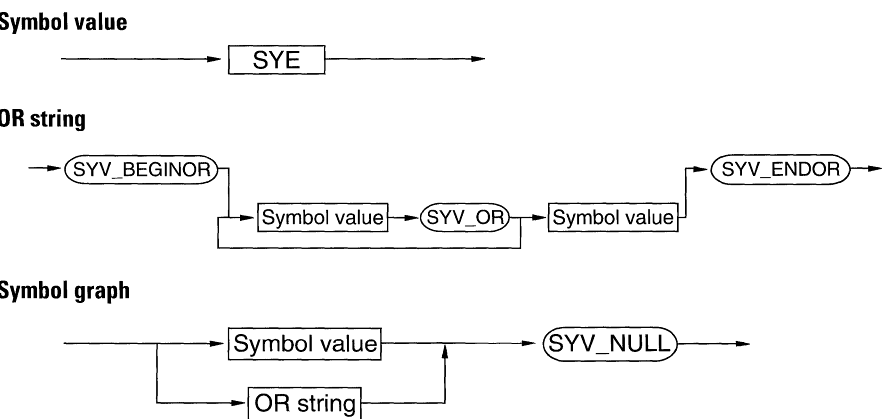



Figure 10.5 Symbol values and symbol graphs

# Pen Messages and Constants

This chapter describes all of the messages and constants used by the Pen API, listed alphabetically. Each entry includes a complete description of the message or constant.

Chapter 9, "Pen API Reference," describes all of the Pen API functions.

Chapter 10, "Pen Structures," describes the structures used in pen computing.

# ALC_Values (Alphabet Code)

```
rc.alc typedef LONG ALC;   // Alphabet code.
```

To enable a subset of the active character set, the application can use OR operators with any of the following values and set the result in the **alc** field of the **RC** structure. The actual characters enabled depend on the language and the **wIntlPreferences** specified in the **RC** structure. For example, if the user has requested French language support, "è" would be included in the lowercase alphabet. Likewise, "£" is included in place of "$" on British systems.

| Value | Definition |
|---|---|
| ALC_ALL | All text written (excludes ALC_USEBITMAP and ALC_RESERVED). |
| ALC_ALPHA | Alphabetic (both lower and uppercase). |
| ALC_ALPHANUMERIC | Alphanumeric (letters and numbers). |
| ALC_DBCS | Returns characters in the double byte encoding (that is, UNICODE). |
| ALC_DEFAULT | Default value; uses complete set of recognizable characters 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.<br><br>If an application sets **rc.alc** to ALC_DEFAULT, and the recognizer is an alphanumeric system recognizer, it 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. |
| ALC_GESTURE | Gesture glyphs. |
| ALC_HIRAGANA | Enables Hiragana characters. |
| ALC_KANJI | Enables Kanji characters. |
| ALC_KATAKANA | Enables Katakana characters. |
| ALC_LCALPHA | Lowercase alphabetic. |
| ALC_MATH | The following characters:<br>% ^ * ( ) - + = { } < > , / . |
| ALC_MONETARY | , (comma), . (period), and $ (dollar sign). Optionally, the monetary characters can vary according to the recognizer and language setting to include, for example, the symbols ¢ (cent sign), ¥ (yen sign), and £ (pound sterling). |

<table>
<tr><th>Value</th><th>Definition</th></tr>
<tr><td>ALC_NONPRINT</td><td>Nonprinting characters: space, tab, return, control characters, glyphs.</td></tr>
<tr><td>ALC_NOPRIORITY</td><td>The default value for rc.alcPriority. This value means the application has no preference for one type of symbol over another.</td></tr>
<tr><td>ALC_NUMERIC</td><td>0 through 9.</td></tr>
<tr><td>ALC_OEM</td><td>Bits reserved for recognizer capabilities specific to the original equipment manufacturer (OEM).</td></tr>
<tr><td>ALC_OTHER</td><td>@ # | _ (underline) ~ (tilde) and square brackets [ ]. That is, all other symbols not included in ALC_ALPHANUMERIC, ALC_MONETARY, ALC_MATH, and ALC_PUNC.</td></tr>
<tr><td>ALC_PUNC</td><td>The following characters: \ ! - ; ‘ “ ? ( ) & . , ;</td></tr>
<tr><td>ALC_RESERVED</td><td>Reserved for future use.</td></tr>
<tr><td>ALC_SYSMINIMUM</td><td>Minimum set of characters needed for Roman alphabet system recognizers: ALC_ALPHANUMERIC | ALC_PUNC | ALC_WHITE | ALC_GESTURE.</td></tr>
<tr><td>ALC_UCALPHA</td><td>Uppercase alphabetic.</td></tr>
<tr><td>ALC_USEBITMAP</td><td>See the explanation following this table.</td></tr>
<tr><td>ALC_WHITE</td><td>White space. If this value is not included in the rc.alc field, 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.</td></tr>
</table>

If ALC_USEBITMAP is set, the 256-bit bitfield in **rc.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 with any characters enabled, using the other ALC_ codes. A “1” set in a bit array indicates that the character is enabled.

If the character enabled in the **rc.rgbfAlc** bitfield is a foreign character, you should also specify the language of the character in the **rc.lpLanguage** field.

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_DCBS. These effects are described in the following table.

| Character in | ALC_DBCS = 0 | ALC_DBCS = 1 |
|---|---|---|
| ALC_HIRAGANA | N/A | 0x82a0 - 0x82f2 |
| ALC_KATAKANA | 0xa1 - 0xdf | 0x8340 - 0x8396 |
| ALC_KANJI | N/A | Kanji character (Shift JIS Level 1 Kanji) |

A recognizer will 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 by inserting a special string in the .RC file's CONTROL statement. This string is of the form "ALC<xxxx>" where xxxx represents a case-independent hexadecimal ALC code, without any preceding 0x qualifier. You can append normal window text after this ALC entry.

The following code sample is an example of this entry:

```
CONTROL "ALC<402c>Dollars", IDD_PAID, "hedit", ES_LEFT | ... etc
```

The ALC<402c> value will be stripped out, and Dollars will be left as window text. In this example, 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; there is no initial window text specified.

```
CONTROL "ALC<74000>", IDD_J, "hedit", ES_LEFT | ... etc
```

# BXD_Values (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 10, "Pen Structures."

The following table lists the BXD_ values.

| Constant | Value | Definition |
|---|---|---|
| BXD_CELLWIDTH | 11 | Initial value for **cxBox** in **GUIDE** structure after conversion from dialog units to pixels |
| BXD_CELLHEIGHT | 15 | Initial value for **cyBox** in **GUIDE** structure after conversion from dialog units to pixels |
| BXD_BASEHEIGHT | 12 | 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_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 |

# HN_ Notification Messages

The parent window of an hedit or bedit window receives the same notification messages (EN_*) as the parent of an edit window. The parent window receives a WM_COMMAND message. The *wParam* parameter contains the control ID. The *lParam* parameter contains the edit window handle in its low-order word and the message ID in the high-order word.

The following table lists notification messages specific to hedit and bedit controls.

| Notification Message | Meaning |
|---|---|
| HN_DELAYEDRECOGFAIL | Delayed recognition has failed. The attempted recognition was initiated by an application through an HE_STOPINKMODE message or by the user tapping on a control. See also HE_GETRCRESULTCODE in the table of WM_HEDITCTL messages. |
| HN_ENDREC | The current recognition context has closed. The call to the recognizer for recognition has terminated. |
| HN_RCRESULT | The hedit or bedit control has received a WM_RCRESULT message from the recognizer. |

HN_RCRESULT is sent within a WM_COMMAND message to the parent application after the dictionary processing of the SYG. Any result returned for HN_RCRESULT by the application is ignored. The application is free to modify the RCRESULT to force an end to the processing or some other action.

The application can retrieve the actual pointer to the RCRESULT using:

```
lprcr = (LPRCRESULT) SendMessage( hwndHedit, WM_HEDITCTL,
            HE_GETRCRESULT, 0L );
```

This pointer is valid only during the processing of the HN_RCRESULT message. It is also the actual RCRESULT—not a copy—so any changes to it affect the hedit or bedit control.

# IDC_ Values (Display Cursor)

Any pen display driver must define the following new cursor types.

| Name | Value | Description |
|---|---|---|
| IDC_PEN | 32631 | Default pen. Pen points in the northwest compass direction. |
| IDC_ALTSELECT | 32501 | Upside-down standard arrow used for tap-and-hold selection. |

You can access the tap-and-hold cursor with the following code:

```
HANDLE hPenDLL = GetSystemMetrics( SM_PENWINDOW);
if (hPenDLL)
  SetCursor( LoadCursor( hPenDLL, IDC_ALTSELECT ) );
```

You can also preload the handle and use it instead of calling **LoadCursor** every time.

# PCM_ Values (Pen Collection Mode Values)

Pen collection can be stopped on any of the following conditions set by the PCM_ values in the **lPcm** field of the **RC** structure.

| Value | Meaning |
|---|---|
| PCM_ADDDEFAULTS | If the PCM_ADDDEFAULTS flag is set in **rc.lPcm**, the values of **rc.lPcm** are combined with the OR operators to the **rcGlobal.Pcm** values at the time the recognizer is called. |
| PCM_INVERT | Stops recognition on pen inversion (a pen tap with the blunt end of the pen). |
| PCM_PENUP | Stops when the pen leaves the tablet contact. |
| PCM_RANGE | Stops when the pen leaves the tablet proximity range. |
| PCM_RECTBOUND | Stops when the pen is placed down outside the inclusion rectangle. The inclusion rectangle is specified in the **rectBound** field of the **RC** structure. |
| PCM_RECTEXCLUDE | Stops when the pen is placed down inside the exclusion rectangle. The exclusion rectangle is specified in the **rectExclude** field of the **RC** structure. |
| PCM_TIMEOUT | Stops at time-out. The time-out value is specified in the **wTimeOut** field of the **RC** structure. |

# PDC_ Values (Pen Device Capabilities)

The following table lists the values for the **lPdc** field of the **PENINFO** structure.

| Value | Meaning |
|---|---|
| PDC_BARREL1 | Barrel button 1 is present. |
| PDC_BARREL2 | Barrel button 2 is present. |
| PDC_BARREL3 | Barrel button 3 is present. |
| PDC_INTEGRATED | The display and the digitizer share the same surface. |
| PDC_INVERT | The pen can detect that the other end of the pen is in contact with the tablet. |
| PDC_PROXIMITY | The pen position can be detected even when the tip is not in contact with the tablet. |
| PDC_RANGE | The pen can generate an event when it leaves or enters the proximity range. |
| PDC_RELATIVE | The pen driver can generate only relative coordinates. |

For additional details, see the entry for the **PENINFO** structure in Chapter 10, "Pen Structures."

# PDK_ Values (Pen Driver State Bits)

The pen driver passes along information that the mouse event is being generated by a pen driver, as well as the current state of any barrel buttons. This information is passed along in the **wPdk** field of the **PENPACKET** and **STROKEINFO** structure. The following table lists the PDK_ values.

| Constant | Value | Meaning |
|---|---|---|
| PDK_BARREL1 | 0x0002 | Set if barrel button 1 is depressed. |
| PDK_BARREL2 | 0x0004 | Set if barrel button 2 is depressed. |
| PDK_BARREL3 | 0x0008 | Set if barrel button 3 is depressed. |
| PDK_DOWN | 0x0001 | Set if the pen is in contact with the tablet. |
| PDK_DRIVER | 0x8000 | Set if generated by a pen driver (as opposed to a mouse driver). |
| PDK_INVERTED | 0x0080 | Set if the other end of the pen is being used as the tip. |
| PDK_OUTOFRANGE | 0x4000 | Set if the driver detects the pen leaving the range of detection. If set, other information in the packet is invalid. |
| PDK_TRANSITION | 0x0010 | Has meaning only if set by **GetPenHwData**. 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 by **GetPenHwData**.<br><br>If this bit is set on a call to **AddPointsPenData,** a new stroke will be created even if the previous call to **AddPointsPenData** appended points of the same pen state. By default, a contiguous 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. |

Bits 8 through 11 are used to indicate which physical pen generated the event. Pens are numbered starting at 0. Other bits are reserved.

# PDT_ Values (OEM-Specific Data)

| Constant | Value |
|---|---|
| PDT_NULL | 0 |
| PDT_PRESSURE | 1 |
| PDT_HEIGHT | 2 |
| PDT_ANGLEXY | 3 |
| PDT_ANGLEZ | 4 |
| PDT_BARRELROTATION | 5 |
| PDT_OEMSPECIFIC | 16 |

For additional information, see the **PENINFO** and **OEMPENINFO** structures in Chapter 10, "Pen Structures."

# PDTS_ Values (Data Scaling Values)

| Constant | Value |
|---|---|
| PDTS_ARBITRARY | The application has performed its own scaling of the data point. |
| PDTS_DISPLAY | Each logical unit is mapped 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 mm. Positive x is to the right; positive y is down. |
| PDTS_LOMETRIC | Each logical unit is mapped to 0.01 mm. Positive x is to the right; positive y is down. |
| PDTS_STANDARDSCALE | The standard scaling metric is equivalent to PDTS_HIENGLISH. |
| PDTS_TABLET | The data points are in tablet units. |

For additional information, see the entries for the **CompactPenData** and **MetricScalePenData** functions in Chapter 9, "Pen API Reference."

# RCD_ Values (Writing Direction)

```
rc.wRcDirect;
```

The **wRcDirect** field informs the recognizer of the direction of writing. 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).

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.

| Value | Meaning |
|---|---|
| RCD_DEFAULT | Default value |
| RCD_BT | Bottom to top |
| RCD_LR | Left to right |
| RCD_RL | Right to left |
| RCD_TB | Top to bottom |

For example, the value for normal English writing direction is defined as follows:

```
#define wRcDirectRoman  ((RCD_LR<<8) | RCD_TB)
```

# RCO_Values (Recognition Option)

```
rc.lRcOptions;
```

This field specifies various options for recognition. It is a logical OR of any of the following values. The following table lists the RCO_ values.

| Value | Meaning |
|---|---|
| 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 the question-mark 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 passes on all data to the recognizer. This is useful, for example, if the application has installed a shape recognizer so the user can enter dots of ink.<br><br>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 and optional data such as pressure. By default, only data used by the recognizer is saved.<br><br>The Microsoft recognizer collects all data from first to last downstroke, including upstrokes 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. |

| Value | Meaning |
|---|---|
| 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 still no string is 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 fields representing coordinate values in the **RC** structure are in tablet coordinates instead of screen coordinates. This can be used to collect recognition data on the portion of the tablet not mapped to the screen. |

# RCOR_Values (Tablet Orientation)

```
rc.wRcOrient;
```

The **wRcOrient** field specifies the orientation of the tablet. Orientation should not be confused with direction of writing, which is described later in this chapter. Orientation can be useful in a charting program that allows the user to label the vertical axis.

The orientation does not affect the raw data that is passed back in the **RCRESULT** structure. Internally to the recognizer, however, the orientation is used to direct the transformation of tablet coordinates to ideal coordinates used for recognition. 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.

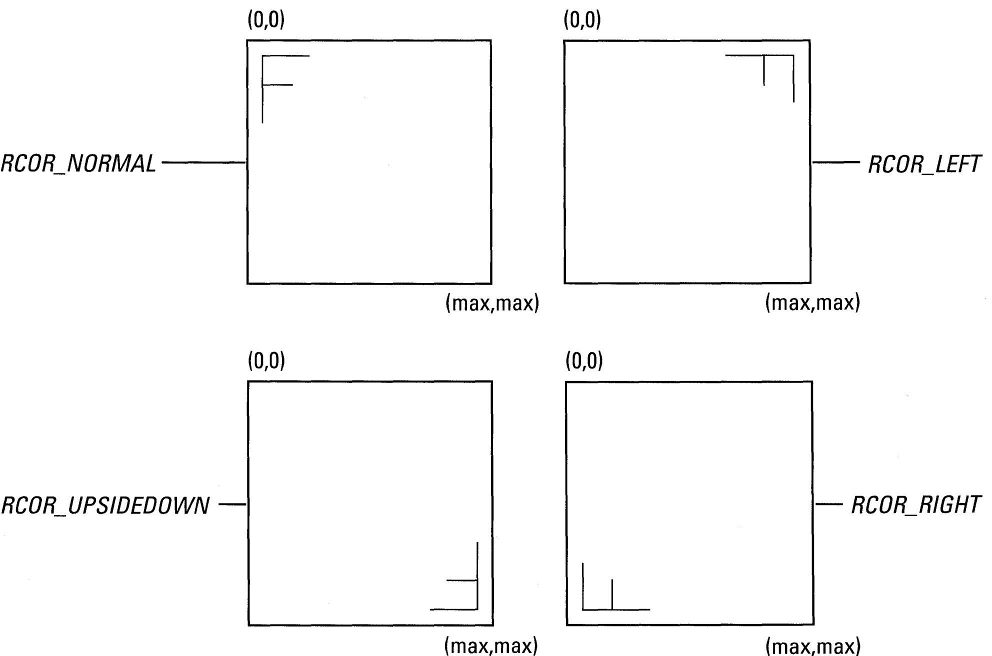



Figure 11.1. Tablet orientation

# RCP_Values (User Preferences)

```
WORD wRcPreferences;
```

The **wRcPreferences** field specifies the user preferences as a combination of RCP_ constants, as described in the following table.

| Constant | Meaning |
|---|---|
| RCP_LEFTHAND | User writes with left hand. |
| RCP_MAPCHAR | Tells the recognizer to fill in segmentation information in the **lpsyc** field. This value cannot be set by the user (there is no Control Panel access to it). RCP_MAPCHAR is used by the Trainer. |

# RCRT_Values (Results Type)

```
rcresult.wResultsType
```

The **wResultsType** field can contain any of the following values.

| Value | Meaning |
|---|---|
| RCRT_ALREADYPROCESSED | Set by a hook or the Gesture Manager 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 or the Gesture Manager can set this flag. The Hform sample application demonstrates its use. |
| RCRT_GESTURE | Result is a gesture symbol. |
| RCRT_GESTURETOKEYS | Gesture Manager translated the gesture to a set of virtual keys. |
| RCRT_GESTURETRANSLATED | Gesture Manager translated the gesture to an ANSI text value. |
| 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. |

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 chracter
          if (lpr->wResultsType & RCRT_GESTURE)
            {
                    // Handle Gesture
                ... code ...
            }
```

```
                else
                {
                        // Character results
                     ... code ...
                }
              }
            else
              {
                     // Handle special cases as necessary.  In general,
                     // should just ignore.  This is what hedits do.
                     ... code ...
              }
}
```

# REC_ Values (Recognition Functions)

The following return values are used as return values from **Recognize**, **RecognizeData**, **ProcessWriting**, **GetPenHwEventData**, and **GetPenHwData.** They are also returned as the *wParam* value of the WM_RCRESULT message. Return values of less than REC_DEBUG are provided for debugging purposes only and represent abnormal termination.

| Value | Meaning |
|---|---|
| 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* in *lprc* 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_NOINPUT | Returned by **RecognizeData** if the buffer contains no data, or returned by **Recognize** if recognition ended before any data is collected. For example, a pen down occurs outside the bounding rectangle before any data is 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 holds the position for a short period of time.<br><br>REC_POINTEREVENT indicates that the application should begin selection actions rather than inking or recognition. If this is returned no WM_RCRESULT message is generated and no ink will be displayed. |

| Value | Meaning |
|---|---|
| REC_TERMBOUND | Recognition ended because of a hit test outside the bounding rectangle. The **rcresult.pntEnd** field is filled with the point causing the stop. |
| REC_TERMEX | Recognition ended because of a hit test inside the exclusion rectangle. The **rcresult.pntEnd** field is filled with the point causing the stop. |
| REC_TERMOEM | Values >= 512 reserved for recognizer-specific termination reasons. |
| REC_TERMPENUP | Recognition ended on pen up. The **rcresult.pntEnd** field 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 continuously for a given amount of time.) |

## Debugging Values

All of the values listed in the following table are in the debug version only. No WM_RCRESULT message was generated if these values were returned by **Recognize**.

| Value | Meaning |
|---|---|
| REC_DEBUG | All debugging return values are less than or equal to this. |
| REC_ALC | Invalid enabled alphabet. |
| REC_BADEVENTREF | Returned when the **wEventRef** field in the *lprc* 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. |

| Value | Meaning |
|---|---|
| REC_LANGUAGE | Returned by the recognizer when the **lpLanguage** field 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 this (below –1024) are specific to the recognizer. |
| REC_PCM | Invalid *lPcm* parameter. There is no way for the recognition to end. |
| REC_RECTBOUND | Invalid rectangle. |
| REC_RECTEXCLUDE | Invalid rectangle. |
| REC_RESULTMODE | Unsupported results mode requested. |

# SYV_ Values (Symbol)

```
typedef LONG SYV;    // Symbol value
```

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 JIS Level 1 (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

| Value | Meaning |
|---|---|
| SYV_BEGINOR | Begins a list of choices; in this document, displayed as an opening brace character: { |
| SYV_EMPTY | Empty. |
| SYV_ENDOR | Ends a list of choices. In this document, displayed as a closing brace character: } |
| SYV_NULL | NULL terminator. |
| SYV_OR | Separator for list of choices; in this document, displayed as a vertical bar: \| |

| Value | Meaning |
|---|---|
| SYV_SOFTNEWLINE | Translated to a space by the **SymbolToCharacter** function. When breaking strings into words, **DictionarySearch** treats SYV_SOFTNEWLINE as a space. |
| 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.

The Win 3 Equivalent column shows the Windows 3.0 mouse and keyboard equivalents.

| Name | Value | Description | Win 3 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_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. | |
| 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_PASTE | 0x0002FFDC | Pastes selection at point indicated by hotspot of paste gesture. | Click (place insertion point) followed by SHIFT+INS |
| SYV_RETURN | 0x0002000D | Enters RETURN key. | Click, RETURN |

| Name | Value | Description | Win 3 Equivalent |
|---|---|---|---|
| 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. | |

## Circle Gesture Symbol Values

The circle gestures consist of the alphabetic characters surrounded, in each instance, by a circle. The characters can be uppercase or lowercase letters. Gestures can be mapped to specific user actions by means of the Gesture Manager.

The following table lists the SYV_ values for the circle gestures. The intervening values correspond to the letters between "a" and "z":

| Constant | Value | Meaning |
|---|---|---|
| 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 |

# WM_GLOBALRCCHANGE Message

Whenever a change is made to the global **RC** values, the WM_GLOBALRCCHANGE message is sent to all top-level windows. The *wParam* and *lParam* values are not used; they are set to zero.

When an application receives a WM_GLOBALRCCHANGE message, it should call **ConfigRecognizer** with a WCR_RCCHANGE subfunction request for all recognizers the application has loaded (excluding the default recognizer).

Similarly, an application should call **DictionaryProc** with a DIRQ_RCCHANGE subfunction request for all dictionaries the application has loaded (excluding the default dictionaries).

For more details, see the entries for **ConfigRecognizer** and **DictionaryProc** in Chapter 9, "Pen API Reference."

# WM_HEDITCTL Messages

Any control message (EM_*) that can be sent to an edit control can also be sent to an hedit window. Most EM_* control messages are also supported by bedit controls, except as noted in the following table. In addition, a single new message, WM_HEDITCTL, has been added for hedit and bedit controls.:

```
lRet = SendMessage(hwndEdit, WM_HEDITCTL, HE_xxx, lParam);
```

The message function is indexed by *wParam,* the values by *lParam.* The following table gives the different values for *wParam* and *lParam*, as well as the return values. All HE_ messages are common to both hedit and bedit controls except as noted. In a bedit control, each cell contains one logical character. In a bedit control, carriage return (CR) and line-feed (LF) bytes together form one logical character.

| wParam value | lParam value | Returns (BOOL) |
|---|---|---|
| HE_CHAROFFSET (bedit only) | Converts logical character position of a character in the control to byte offset to the character. Both the logical character position and the byte offset are zero-based. The LOWORD contains the logical character position. The HIWORD is reserved and must be set to zero. | If the supplied logical character position is less than the total number of logical characters in the control, the LOWORD of the return value contains the byte offset and the HIWORD is zero. Otherwise, the LOWORD contains the length of text in bytes and the HIWORD contains 0xFFFF.<br><br>See the related HE_CHAROFFSET. |
| HE_CHARPOSITION (bedit only) | Converts byte offset in the text buffer of the control to the logical character position, which contains the byte specified by the byte offset. Both the byte offset and the logical character position are zero-based. The LOWORD contains the byte offset. The HIWORD is reserved and must be set to zero. | If the supplied byte offset is less than the length of the text in bytes, the LOWORD of the return value contains the logical character position, and the HIWORD is zero. Otherwise, the LOWORD contains the total number of logical characters in the text of the control, and the HIWORD contains 0xFFFF.<br><br>See the related HE_CHAROFFSET. |

| wParam value | lParam value | Returns (BOOL) |
|---|---|---|
| HE_DEFAULTFONT (bedit only) | Switches the font of the bedit control to the default font that bedit selects at the time of creation. If the LOWORD of *lParam* is nonzero, the control is repainted. | Undefined. |
| HE_GETBOXLAYOUT (bedit only) | Points to the **BOXLAYOUT** structure, which is filled with the current **BOXLAYOUT** for the control. | Undefined. |
| HE_GETINFLATE | **LPRECTOFS** filled with current value. | TRUE if successful. |
| HE_GETINKHANDLE | Unused. | The LOWORD contains a handle to the captured ink. If NULL, the control is not in ink mode. Applications need to duplicate this handle, because it is no longer valid after the control is destroyed. |
| HE_GETRC | **LPRC** Pointer to the **RC** structure to fill with current values. | TRUE if successful. |

| wParam value | lParam value | Returns (BOOL) |
|---|---|---|
| HE_GETRCRESULT | Unused. | Pointer to an **RCRESULT** structure received by the control. This message can be sent only during the processing of an HN_RCRESULT notification.<br><br>Any modifications the application makes to the **RCRESULT** it receives directly affect the **RCRESULT** used by the control. The pointer to **RCRESULT** is valid only during the processing of the HN_RCRESULT notification. |
| HE_GETRCRESULTCODE | Unused. | Returns the value returned by the last delayed recognition. It can be called only in response to an HN_ENDREC, an HN_RCRESULT, or an HN_DELAYEDRECOGFAIL notification. |
| HE_GETUNDERLINE (hedit only) | Unused. | TRUE if underline mode is set. |
| HE_SETBOXLAYOUT (bedit only) | Points to the **BOXLAYOUT** structure to be set. | TRUE if successful. |
| HE_SETINFLATE | **LPRECTOFS** specifies the adjustments to the client rectangle of the control window to specify the size of the writing window. | TRUE if successful, FALSE if an invalid window rectangle is specified. |
| HE_SETINKMODE | Starts the collection of inking. The LOWORD is the initial **HPENDATA**. It can be NULL. If initial **HPENDATA** is supplied, it must be relative to the top-left corner of the client rectangle of the control. | TRUE if successful. |

| wParam value | lParam value | Returns (BOOL) |
|---|---|---|
| HE_SETRC | **LPRC** pointer to the **RC** structure to set. | TRUE if successful. For details about the fields ignored or overridden in the **RC**, see the discussion following this table. |
| HE_SETUNDERLINE (hedit only) | The LOWORD is TRUE to set the underline and FALSE to reset it.<br>Note that to use underline mode with hedit controls, the WS_BORDER style bit must be turned off. | The current underline mode. |
| HE_STOPINKMODE | Stops the collection of ink. If the LOWORD is HEP_RECOG, it performs recognition and displays text. If the LOWORD is HEP_NORECOG (zero), it removes the ink without performing recognition. If the LOWORD is HEP_WAITFORTAP, it performs recognition on the next tap in the control. | TRUE if successful. |

Before using the HE_SETBOXLAYOUT, HE_SETINFLATE, or HE_SETRC *wParam* values, it is often useful to retrieve the current structure associated with the control using the HE_GETBOXLAYOUT, HE_GETINFLATE, or HE_GETRC *wParam* values. You should then change the fields of interest in the retrieved structure. This reduces the risk of inadvertent changes in the fields of the structure that are not of interest.

Note that, with the HE_SETRC *wParam* value, certain fields in the **RC** are overridden or ignored in hedit and bedit controls. For the following fields, values are calculated by the control when necessary; when set by the application with a call to HE_SETRC, however, the values for these fields are ignored:

- **rc.hwnd**
- **rc.rectBound**
- **rc.wResultMode**
- **rc.wEventRef**
- **rc.lRcOptions**
- **rc.lPcm**

With **rc.lRcOptions**, the RCO_TABLETCOORD is always forced off. If the application sets RCO_SAVEHPENDATA, it must process the HN_RCRESULT to get the result and free the *hpendata.*

With **rc.lPcm**, the PCM_RECTBOUND is always set, and PCM_RECTEXCLUDE is always cleared.

In addition, for bedit controls the values set by the user for the following fields are ignored or overridden:

- **rc.guide.xOrigin**
- **rc.guide.yOrigin**
- **rc.guide.cHorzBox**
- **rc.guide.cVertBox**
- **rc.lRcOptions**
- **rc.wRcOrient**

Note that with **rc.lRcOptions**, RCO_BOXED is always forced on. With **rc.wRcOrient**, only RCOR_NORMAL is supported.

- rc.wResultMode

Using the RRM_SYMBOL value for **wResultMode** in bedit controls disables all default dictionary processing. An application may perform dictionary processing on its own by getting the recognition results during the processing of HN_RCRESULT notification and calling the **DictionarySearch** function.

# WM_RCRESULT Message

The WM_RCRESULT message is generated as a result of a call to **ProcessWriting**, **Recognize**, or **RecognizeData**.

The *wParam* parameter of the message contains the reason the recognition ended (one of the REC_ codes). It is REC_OK if more results are to be sent; otherwise, it is the same value returned by **Recognize** or **RecognizeData**. The *lParam* parameter is a far pointer to an **RCRESULT** structure.

If the input consists of multiple lines, **nBaseLine** and **nMidLine** in the **RCRESULT** structure are set to the value for the first line. If the input has a vertical writing direction, the value of **nBaseLine** represents the left alignment edge.

Any far pointers passed in the **RCRESULT** structure are valid only while processing the message. This is the application's chance to save the information about the raw data. After this message is sent, the recognizer is free to destroy its copy of the raw data.

The last WM_RCRESULT message for a recognition context is sent before **Recognize** returns and any other messages are sent to the application.

If the application returns 1, the RC Manager should perform no further processing. If the application returns REC_ABORT, the recognition is ended; no more results will be sent for this context. If the application returns 0, it means the application did not process the RCRESULT message; the RC Manager should perform any further default processing. This distinction is relevant only to **ProcessWriting**.

# WM_SKB Message

The WM_SKB message is posted to all windows when the on-screen keyboard changes (for example, position, visibility, keypad visibility, restored or minimized changes). Clients and other applications can use this message to monitor its state.

The *wParam* value is SKN_CHANGED, and the LOWORD of *lParam* is a combination of one or more of the following values listed below. The HIWORD value is the window handle of the keyboard.

| Value | Meaning |
|---|---|
| SKN_MINCHANGED | The on-screen keyboard has been minimized or restored. |
| SKN_PADCHANGED | The keypad display of the on-screen keyboard has changed. |
| SKN_POSCHANGED | The position of the on-screen keyboard has changed. |
| SKN_VISCHANGED | The on-screen keyboard has been shown or hidden. |

# Appendix A: Guide to the Initialization Files

This appendix describes the settings used in the SYSTEM.INI, WIN.INI, CONTROL.INI, and PENWIN.INI initialization files and how to change them.

The SYSTEM.INI file is one of two Windows initialization files that contain information used by the Microsoft Pen Extensions; the other is WIN.INI. Both are included with MS Windows.

The PENWIN.INI file is an initialization file used for pen computing. It contains information that defines your pen and handwriting environment.

For information on general Windows settings in the SYSTEM.INI or WIN.INI file, see your MS Windows documentation.

All initialization files reside in the directory where Windows is installed.

The Control Panel provides user access to the commonly needed entries in the initialization files. In general, a user of the Pen Extensions should never need to inspect or modify the .INI file entries directly.

**Note** If you change a setting incorrectly in SYSTEM.INI, you can disable your system.

## FORMAT of .INI files

All Windows initialization files have the following format:

```
[section name]
keyname=value
keyname=value
   .
   . other keynames go here
   .
[section name]
keyname=value
keyname=value      ; comments may also appear on a regular line
; comments are preceded by semicolons
```

The [section name] parameter is the name of a section. Sections are used to break settings into logical groups. The enclosing brackets [ ] are required, and the left bracket must be in the leftmost column on the screen.

The `keyname=value` statement defines the value of each setting. A keyname is the name of a setting. It can consist of any combination of letters and digits, and must be followed immediately by an equal sign (=). The value of the setting can be an integer, a Boolean value, a string, or a quoted string, depending on the setting. There are multiple settings in most sections.

You can include comments in initialization files. You must begin each line of comments with a semicolon (;).

This appendix assumes that you have installed the Microsoft Windows for Pen Computing SDK into the C:\PENSDK directory. If you have installed it in some other directory or on a different drive, you must make substitutions to the SYSTEM.INI, WIN.INI, and PENWIN.INI files where appropriate.

# Modifying the SYSTEM.INI File

You can use either the mouse or a digitizing tablet as your input device. You must modify your SYSTEM.INI file depending on which device you will be using.

## Mouse Input Device

If you are using the mouse as your input device, make the following entries to your SYSTEM.INI file.

| SYSTEM.INI entry | Description |
|---|---|
| `[boot]` | Section name |
| `mouse.drv=c:\pensdk\bin\yesmouse.drv` | Custom mouse driver that simply displays the cursor |
| `display.drv=c:\pensdk\bin\vga.drv` | Sets video driver that supports inking |
| `drivers=pen penwindows` | Defines installable drivers |
| `[drivers]` | Section name |
| `penwindows=c:\pensdk\bin\penwin.dll` | Sets the Microsoft Windows for Pen Computing system |
| `pen=c:\pensdk\bin\msmouse.drv` | Sets installable pen driver using mouse |
| `[boot.description]` | Section name |
| `display.drv=VGA for Pen` | Name change |

## Pen Input Device

If you are using a pen tablet instead of the mouse, use the following SYSTEM.INI entries rather than those listed above.

| SYSTEM.INI entry | Description |
|---|---|
| `[boot]` | Section name. |
| `mouse.drv=mouse.drv` | Set back to the default setting. |
| `display.drv=c:\pensdk\bin\vga.drv` | Sets video driver that supports inking. |
| `drivers=pen penwindows` | Defines installable drivers. |

| SYSTEM.INI entry | Description |
|---|---|
| `[drivers]` | Section name. |
| `pen=c:\pensdk\bin\wacom.drv` | Sets the pen driver for the Wacom tablet. If you are using a different tablet, substitute it here for WACOM.DRV. |
| `penwindows=c:\pensdk\bin\penwin.dll` | Sets the MS Windows for Pen Computing system. |
| `[boot.description]` | Section name. |
| `display.drv=VGA for Pen` | Name change. |
| `[386Enh]` | Section name. |
| `device=c:\pensdk\bin\wacom.386` | Sets the device driver for the Wacom tablet. If you are using a different tablet, replace this line with the appropriate .386 name. |
| `[Pen driver]` | Section name. |
| | The Wacom serial driver also supports additional SYSTEM.INI settings. |
| `com2=1` | Set to 1 if the tablet is connected to COM2. |
| `wacom510=1` | Set if the tablet is the opaque HD-510 tablet. |
| `pressure=1` | Set if the tablet uses pressure-sensitive pen. |
| `inductive=1` | Set if the tablet uses an inductive pressure pen. |

In general, you should use VGAMONO.DRV in place of VGA.DRV. The monochrome VGA display driver gives better display results on a one-plane integrated tablet.

# Pen Display Orientation

The following comments assume you are using a tablet or display that supports different orientations.

| SYSTEM.INI entry | Description |
|---|---|
| `[Display Driver]` | Section name. |
| `DisplayOrientation=`*n* | Specifies the tablet orientation. The value of *n* is the number of times the screen and tablet have been rotated 90 degrees counter-clockwise (default is 0).<br>The ROTATE.DRV display driver can rotate the screen only 90 degrees counterclockwise. The only valid values with this driver are 0 or 1. |
| `OrientableDrivers=c:\pensdk\bin\vga.drv, c:\pensdk\bin\rotate.drv` | Specifies, on a single line, the path to the rotatable screen drivers. This entry can have more than two items listed. They should all be listed on one line. The format is: OrientableDrivers=device driver 1, driver 2, driver 3, driver 4<br>Each driver in the line represents the driver for 0, 90, 180, and 270 degrees rotation right, respectively. |

# Modifying the CONTROL.INI File

The following table contains a description of the entries related to pen computing in the CONTROL.INI initialization file.These changes are not required if the *.CPL files from C:\PENSDK\BIN are copied to the SYSTEM subdirectory of the directory where Windows was installed.

| CONTROL.INI Entry | Description |
|---|---|
| `[MMCPL]` | Section name. |
| `cppen=c:\pensdk\bin\cppen.cpl` | Adds the pen properties customization item (Pen) to the Control Panel. |
| `cphw=c:\pensdk\bin\cphw.cpl` | Adds the recognition parameter customization item (Handwriting) to the Control Panel. |
| `cprot=c:\pensdk\bin\cprot.cpl` | Adds the display orientation customization item (Rotate) to the Control Panel. |
| `cpcal=c:\pensdk\bin\cpcal.cpl` | Adds the tablet calibration item (Calibrate) to the Control Panel. |

| CONTROL.INI Entry | Description |
|---|---|
| NumApps=x | The *x* specifies the number of applications in the control panel. After installing the Pen Windows system, it will be 4 larger than the previous value, reflecting the four additional entries noted above. |

# Modifying the PENWIN.INI File

The following table contains a description of the entries in the PENWIN.INI initialization file. Most of these values should not be changed directly. The user should use the Windows Control Panel to change them. Unless specified, numeric values are decimal numbers. A "CPL" following the description indicates that you can use the Control Panel to set the entry.

Current values of several entries in PENWIN.INI are available in the global recognition context. To read or write these entries, an application should use the **GetGlobalRC** and **SetGlobalRC** functions.

| PENWIN.INI Entry | Description |
|---|---|
| [Current] | Section name. This is a required section of PENWIN.INI that defines the current parameters. You set the values with the Control Panel. |
| User=Default User | The user's name. (CPL) |
| InkWidth=1 | Ink width (valid values are 0 through 15). (CPL) |
| InkColor=0000FF | Ink color, in hexadecimal values (valid values: 0 through FFFFFF). (CPL) |
| SelectTimeOut=250 | Delay from pen tap-down time to the beginning of selection mode. The delay is specified in milliseconds. (CPL) |
| [*Default User] | Section name. There will be a section like this for each user on the system. |
| TryDictionary=100 | Specifies the cutoff for the number of enumerations per symbol graph. For more information, see the entry for **rc.TryDictionary** in Chapter 10, "Pen Structures." |

| PENWIN.INI Entry | Description |
|---|---|
| `ErrorLevel=25` | Confidence threshold. The recognizer does not return anything with confidence level below the **ErrorLevel**. For more information, see the entry for **rc.clErrorLevel** in Chapter 10, "Pen Structures." |
| `EndRecognition=8000` | A long hexadecimal value bit field specifying when recognition should end (for example, proximity). For more information, see the entry for **rc.lPcm** in Chapter 10, "Pen Structures." (CPL) |
| `TimeOut=1000` | How long the system waits (in milliseconds) while there is no more pen input before recognition terminates. For more information, see the entry for **rc.wTimeOut** in Chapter 10, "Pen Structures." (CPL) |
| `WriteDirection=103` | The writing direction (in hexadecimal values). For more information, see the entry for **rc.wRcDirect** in Chapter 10, "Pen Structures." |
| `MenuDropAlignment=0` | The menu alignment. 0 indicates a drop to the right; 1 indicates a drop to the left. (CPL) |
| `Preferences=0` | A hexadecimal bit field for recognition context preferences. For more information, see the entry for **rc.wRcPreferences** in Chapter 10, "Pen Structures." (CPL) |
| `IntlPreferences=0` | A hexadecimal bit field for international recognition context preferences. For more information, see the entry for **rc.wRcIntlPreferences** in Chapter 10, "Pen Structures." (CPL) |
| `Recognizer=c:\pensdk\bin\mars.dll` | Specifies the path to the recognizer currently selected by the user. (CPL) |

| PENWIN.INI Entry | Description |
|---|---|
| `[User List]` | Section name that lists the defined users. |
| `Default User=` | If non-zero, denotes the default user for the system. You should always use the Control Panel to set a unique user name. (CPL) |

A series of entries in the PENWIN.INI file is devoted to the dictionaries and recognizers used by the system. The following table describes these entries.

| PENWIN.INI Entry | Description |
|---|---|
| `[Dictionary List]` | Section name. This section contains lists of dictionaries available on the system. You set this only in the PENWIN.INI file.<br>Each entry specifies the path to a dictionary DLL. Setup programs for the dictionaries should add and remove entries in this section. |
| `c:\pensdk\bin\userdict.dll=` | Note: nothing follows the equal sign. |
| `c:\pensdk\bin\maindict.dll=` | Note: nothing follows the equal sign. |
| `[*Default User.Dictionary List]` | Section name that defines the dictionaries currently used by the user "Default User." There will be one such section for each user who uses the default dictionaries. |
| `0=c:\pensdk\bin\userdict.dll` | Dictionaries currently used by the user. The number indicates the search order. (CPL) |
| `1=c:\pensdk\bin\maindict.dll` | (CPL) |
| `[Recognizer List]` | Section name. |
| `c:\pensdk\bin\mars.dll=` | List of recognizers available on the system. Set this only in the PENWIN.INI file.<br>Each entry specifies the path to a recognizer DLL. Setup programs for the recognizers should add and remove entries in this section. |

| PENWIN.INI Entry | Description |
|---|---|
| `[mars]` | Section name for specific Microsoft recognizer. |
| `msdb=c:\pensdk\bin\beta.mob` | The character database against which handwriting input is compared. |
| `[MsMainDict]` | Section name for main and short dictionaries to use. These entries reflect the Microsoft dictionary implementation and will vary if other dictionaries are used. |
| `XXXMain=<path>mssp_yy.lex` | The DLL looks for the keyword `XXXMain=`, where `XXX` is any of the three-letter language codes under the `[MsMainDict]` section in PENWIN.INI to load the language word list. If `XXXMain` is found, the DLL tries to load the files the keyword points to. Otherwise, the DLL looks for `mssp_yy.lex` in the BIN directory of wherever you installed the SDK.<br><br>The "yy" letters stand for one of the two-letter language-specific codes listed in the description for the **DictionaryProc** function.<br><br>The default value is xxx=emu and yy=am. |
| `[MsUserDict]` | Section name for user word lists (optional). |
| `c:\pensdk\bin\list1.dic=` | Default user word lists that the Microsoft User Dictionary DLL searches (optional). |
| `c:\pensdk\bin\list2.dic=` | Same as preceding. |
| `c:\pensdk\bin\list3.dic=` | Same as preceding. |
| `[MsSpell]` | Section name. |
| `c:\pensdk\bin\msspell.dll=` | Default user word lists that the Microsoft User Dictionary DLL searches (optional). |

The Pen Palette uses a section of the PENWIN.INI file to maintain its settings. The following table lists these entries. You can configure any of them with the Pen Palette settings.

| PENWIN.INI Entry | Description |
|---|---|
| [Pen Palette] | Section name for Pen Palette entries. |
| Minimized=1 | Pen Palette status. 1 = iconic; 0 = restored. The Pen Palette is started initially in the minimized, iconic state. |
| WritingWindowOpen=1 | Writing window status. 1 = open; 0 = closed. |
| WindowPos=18 263 260 414 | Position of Pen Palette (left, top, right, bottom), including the writing window. This is measured in screen coordinates. |
| Comb=0 | Letter guide status. 1 = use boxed input; 0 = use a standard hedit window. |
| AutoWrite=1 | Automatic writing status. 1 = use autowriting on text fields; 0 = do not use autowriting. |
| TrainOpen=0 | Training window status. 1 = training window is open; 0 = training window is closed. |
| SKBOpen=1 | On-screen keyboard status. 1 = on-screen keyboard displayed; 0 = on-screen keyboard closed. |
| SKBPos=26 68 | Position of the left, top coordinate of the on-screen keyboard (in screen coordinates). |
| SKBView=3 | On-screen keyboard template setting (101-key, numeric keypad, and so on). |
| SKBInvert=0 | On-screen keyboard color status. 1 = invert all colors of the on-screen keyboard; 0 = use the normal colors. You can set this item only in the PENWIN.INI file. |
| SKBMinimized=0 | On-screen keyboard status. 1 = minimized; 0 = restored. |
| TrainPos=321 297 | Position of the left, top coordinate of the trainer (in screen coordinates). |
| TrainOpen=0 | Trainer status. 1 = displayed; 0 = closed. |
| TrainMinimize=0 | Trainer status. 1 = minimized; 0 = restored. |
| DisableTrain=0 | Trainer status. 0 = Trainer enabled; 1 = Trainer disabled. |

# Modifying the WIN.INI File

The following comments assume you are using a tablet or display that supports different orientations.

## Pen Display Orientation

| WIN.INI entry | Description |
|---|---|
| `[Display Driver]` | Section name. |
| `DisplayOrientation=`*n* | Specifies the tablet orientation. The value of *n* is the number of times the screen and tablet have been rotated 90 degrees counter-clockwise.<br><br>The **DisplayOrientation** field does not affect the display driver at all. The ROTATE.DRV display driver rotates the screen 90 degrees counterclockwise, so people should use **DisplayOrientation=**1 with this display driver. |
| `OrientableDrivers=c:\pensdk\bin\vga.drv, c:\pensdk\bin\rotate.drv` | Specifies the path to the rotatable screen drivers. This entry can have more than 2 items listed. They should all be listed on one line. The format is **OrientableDrivers=***device driver 1, driver 2, driver 3, driver 4*<br><br>Each driver in the line represents the driver for 0, 90, 180, and 270 degrees rotation right, respectively. |

# Appendix B:
# Pen Addenda for MS Windows API Functions

This appendix describes the minor changes and additions made to the MS Windows 3.1 API functions for use with the pen extensions. For information on general Windows API functions, see the *Microsoft Windows Programmer's Reference.*

Each note is listed beneath the name of the Windows API function call.

## GetSystemMetrics

Use the SM_PENWINDOWS index value with **GetSystemMetrics** to retrieve the handle to the penwin DLL. You can use **GetSystemMetrics**(SM_PENWINDOWS) to get the DLL handle to pass to **GetProcAddress**.

The return value will be NULL if Microsoft Windows for Pen Computing is not running.

The following code sample highlights this use:

```
fnRegisterPenApp =
GetProcAddress( (HANDLE) GetSystemMetrics(SM_PENWINDOWS), "RegisterPenApp");
```

## SetClipboardData

Use the CF_PENDATA value with **SetClipboardData** to specify the predefined data format for pen data. Specifically, the CF_PENDATA value is a handle to the pen data memory block (HPENDATA).

## Combo-Box Notification Codes

The following additional notification codes apply to combo boxes used in MS Windows for Pen Computing.

| Message | Description |
|---|---|
| CBN_DELAYEDRECOGFAIL | Sent when delayed recognition has failed |
| CBN_ENDREC | Sent when recognition is ended |
| CBN_RCRESULT | Sent when RCRESULT has been sent |

## Differences Between Bedit and Edit Controls

The parent of the bedit control should not return a nonsolid (patterned) brush to the control in response to the WM_CTLCOLOR message.

The following values have different behaviors if used in bedit or edit controls:

### EM_SCROLL

The EM_SCROLL message is available in bedit controls.

### EM_GETSEL, EM_SETSEL, EM_LIMITTEXT

When these values are used by bedit controls, they are cell indices. When they are used by edit controls, they are byte indices.

### WM_PASTE, EM_REPLACESEL

The bedit control selects the inserted text; edit puts the caret after the inserted text. If there is a single cell selection before the paste operation, bedit does not delete the text in the single cell selection.

## Installable Pen Device Driver

There are no specific API functions for pen driver use. Instead, the pen driver functionality is implemented with installable driver messages.

The pen driver is an installable driver in MS Windows version 3.1. As an installable driver, it may or may not exist. All communication with any installable driver is through driver messages. In order to send an installable driver a message, you need a driver handle. Use the **OpenDriver** Windows version 3.1 function to get the driver handle. For example:

```
HANDLE hDriverPen;
  hDriverPen = OpenDriver( "PenWindows", NULL, NULL );
  if( hDriverPen == NULL )
  {
    /* The pen driver does not exist. */
    /* Either bring up an error message, */
    /* or continue to function as a non-pen-aware application */
  }
```

Once your application has a handle to the installable driver, the application can send the driver messages. For example, 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 information.

```
WORD wOldRate;
wOldRate = SendDriverMessage( hDriverPen, DRV_SetPenSamplingRate,
                              200, NULL);

   .
   .
   .

/* get information about the pen driver */
```

```
PENINFO pi;
BOOL fPenHardwareExists;
fPenHardwareExists = SendDriverMessage( hDriverPen,
          DRV_GetPenInfo,
          (DWORD)(LPPENINFO)&pi,
          NULL );
```

You must close the handle to the installable driver when an application has finished sending messages to the installable driver. For example:

```
/* we won't be sending the driver any more messages now */
CloseDriver(hDriverPen, NULL, NULL);
```

The following table lists the pen driver messages and describes their use.

| Pen Driver Message | Meaning |
|---|---|
| DRV_SetPenDriverEntryPoints | This message is used by the Pen Module after it has been loaded. After receiving this message, the pen driver should call the **OpenDriver** function for the RC Manager and get the address of the appropriate entry points for passing packets into the RC Manager (**AddPenEvent, ProcessPenEvent**). The pen driver must not statically link to the RC Manager or call **OpenDriver** before being called by this function. |
| DRV_RemovePenDriverEntryPoints | This function is used dy the Pen Module just before it is unloaded. Once this message is received, pen drivers should stop calling the **AddPenEvent** and **ProcessPenEvent** entry points in the Pen Module. |
| DRV_SetPenSamplingRate | This message sets the pen sampling rate in samples per second. |
| DRV_SetPenSamplingDist | This message sets the minimum pen sampling distance. Successive points less than the given 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 zero, which means that all pen events generate new events.<br><br>A pen driver does not have to simulate nonzero sampling distances. You need to use the DRV_GetPenInfo driver message to determine the actual sampling distance set.<br><br>A button state transition always generates a new event regardless of the distance of a move. |

| Pen Driver Message | Meaning |
|---|---|
| DRV_GetName | This message reports the name of the pen hardware you're using. Returns TRUE on success. |
| DRV_GetVersion | This message reports the version number of MS Windows for Pen Computing that you're running. |
| DRV_GetPenInfo | This message fills in the **PENINFO** structure pointed to by *lParam1* with the current pen parameters. If this parameter is set to NULL, it checks for the presence of a pen tablet only. |
| DRV_GetCalibration | This message returns the tablet calibration (such as size and offset values). |
| DRV_SetCalibration | This message sets the tablet calibration (such as size and offset values). |

The following table lists the supported driver messages with their parameters and their return values.

| Pen Driver Message | Parameters | Return Value |
|---|---|---|
| DRV_SetPenDriverEntryPoints | lParam1 = 0<br>lParam2 = 0 | None. |
| DRV_RemovePenDriverEntryPoints | lParam1 = 0<br>lParam2 = 0 | None. |
| DRV_SetPenSamplingRate | lParam1 HIWORD = 0<br>lParam1 LOWORD = new sampling rate<br>lParam2 = 0 | HIWORD contains 0; LOWORD contains the sampling rate previously set. |
| DRV_SetPenSamplingDist | lParam1 HIWORD = 0<br>lParam1 LOWORD = new sampling distance<br>lParam2 = 0 | HIWORD contains 0; LOWORD contains the sampling distance previously set. |
| DRV_GetName | lParam1 LOWORD = length of name buffer<br>lParam2 = LPSTR long pointer to the name buffer | Number of characters actually copied. |

| Pen Driver Message | Parameters | Return Value |
|---|---|---|
| DRV_GetVersion | lParam1 = 0<br>lParam2 = 0 | HIWORD contains 0. Within LOWORD, HIBYTE contains the minor version number, LOBYTE the major version number. |
| DRV_GetPenInfo | lParam1 = LPPENINFO (points to a **peninfo** structure to be filled)<br>lParam2 = 0 | HIWORD contains 0; LOWORD contains TRUE if pen hardware exists, FALSE if it does not. |
| DRV_GetCalibration | lParam1 = LPCALBSTRUCT (points to a CALBSTRUCT structure to be filled)<br>lParam2 = 0. | HIWORD contains 0; LOWORD contains 1. |
| DRV_SetCalibration | lParam1 = LPCALBSTRUCT (points to a CALBSTRUCT structure that describes the new calibration parameters the pen driver must use)<br>lParam2 = 0. | HIWORD contains 0; LOWORD contains 1. |

The calibration driver messages use the CALBSTRUCT structure defined below:

```
typedef struct
  {
  int  wOffsetX;
  int  wOffsetY;
  int  wDistinctWidth;
  int  wDistinctHeight;
  } CALBSTRUCT, FAR * LPCALPSTRUCT;
```

The **wOffsetX** and **wOffsetY** fields are the amount in tablet coordinates that need to be added to the x and y values returned by the hardware for proper calibration. The **wDistinctWidth**, and **wDistinctHeight** fields have the same meaning as in the **PENINFO** structure.

# Index

## A

AddPenEvent function 113, 116
AddPointsPenData function 49, 111, 117, 207
alc field 87, 167, 233
ALC_ values
    ALC_ALL 252
    ALC_ALPHA 252
    ALC_ALPHANUMERIC 252
    ALC_DBCS 252
    ALC_DEFAULT 73, 252
    ALC_GESTURE 73, 252
    ALC_HIRAGANA 252, 254
    ALC_KANJI 252, 254
    ALC_KATAKANA 252, 254
    ALC_LCALPHA 252
    ALC_MATH 252
    ALC_MONETARY 252
    ALC_NONPRINT 253
    ALC_NOPRIORITY 253
    ALC_NUMERIC 73, 253
    ALC_OEM 253
    ALC_OTHER 253
    ALC_PUNC 253
    ALC_RESERVED 253
    ALC_SYSMINIMUM 253
    ALC_UCALPHA 253
    ALC_USEBITMAP 253
    ALC_WHITE 253
    description 32
alcPriority field 32, 87, 168,
Alphabet
    definition of in RC structure 233
    enabling 234
    recognition 73
Annotation layer 10
AtomicVirtualEvent function 109, 118

## B

Background processing 34
Barrel buttons 156, 225, 259–60
Bedit
    and edit controls 293–94
    cell sizing 76
    changing standard bedit 218
Bedit *(continued)*
    control 76–77, 277–81
    description 4, 67
    GUIDE data structure 31
    scrolling 76
    specifying characteristics of 216
    switch statement 74
    text wrapping 76
    using 8
BeginEnumStrokes function 48, 111, 119
Bounding rectangles 39, 120, 244
BoundingRectFromPoints function 110, 120
Boxed edit *See* Bedit
BOXLAYOUT data structure
    description 216–17
    sample code 216
Buffer
    and events 33
    and ink display 28
    ANSI string 203
    obtaining data from 159
    pen module control of 112
    pointer to 20
    reasons for 19
Buttons 7
BXD_ values
    BXD_BASEHEIGHT 255
    BXD_BASEHORZ 255
    BXD_CELLHEIGHT 255
    BXD_CELLWIDTH 255
    BXD_CUSPHEIGHT 255
    BXD_ENDCUSPHEIGHT 255
BXS_ values
    BXS_ENDTEXTMARK 216
    BXS_RECT 216

## C

CalcNearestDir function 95–96
Calibration driver messages 299
CBN_ messages
    CBN_DELAYEDRECOGFAIL 295
    CBN_ENDREC 295
    CBN_RCRESULT 295
cbOemData field 44, 227
cbPnts field 246
Character sets 84
CharacterToSymbol function 109–10, 121
cHorzBox field 218, 281
clErrorLevel field 33, 87, 168, 233
CloseRecognizer function 89, 92, 112, 122
Code *See* sample code

Color
GetNearestColor function 29
ink 29, 88, 190, 239
programming considerations 7
setting RC preferences 74
Comb 4
CompactPenData function 50, 111, 123–24
Compressing data 50, 52, 123–24
ConfigRecognizer function 89, 92–93, 112, 125–27, 276
Control
bedit 76–77
ID 76
messages 68, 76
notifications 76
replacing edits with hedits 69
CONTROL.INI file
description 288
sample code 288
Converting ANSI strings 121
Coordinates
tablet 30, 144, 218
tablet to screen 204
values 35
Copying pen data 146
CopyRawData function 60
CorrectWriting function 109, 128–29
Country code 232
cPens field 226
cPnt field 246
CreatePenData function 47, 111, 130–31, 207
CreateWindow function 74
Cursor
copy 34
hiding during inking 34
pen 34
question-mark 34
CUSTDICT.C 100
cVertBox field 218, 281
CWR_ subfunctions
CWR_SINGLELLINEEDIT 129
CWR_STRIPCR 129
CWR_STRIPLF 129
CWR_STRIPTAB 129
CWR_TITLE 129
cxBase field 218
cxBox field 218
cxRawHeight field 226
cxRawWidth field 226
cyBase field 218
cyBox field 218
cyCusp field 216
cyEndCusp field 216


# D


Data
accessing 159, 161–62
buffer 19, 20, 28, 33
compression 50, 123–24, 223
conversion 46, 186–89
copying 146
display 145
ending collection 149
points 117, 175
recognition process 183–85
redraw 190–91
resizing 193
trim options 123–24
Data flow
dictionary processing 22–23, 265
gesture processing 151–52
overview 5, 17–26
recognition 22, 27–40, 81–83
Data structures
pen
SYG, SYE, SYC, & SYC *See* SYG, SYE, SYC, & SYV data structures
adding data points 117
BOXLAYOUT *See* BOXLAYOUT data structure
CALBSTRUCT 299
GUIDE *See* GUIDE data structure
HPENDATA *See* HPENDATA data buffer
OEMPENINFO 82
PENDATAHEADER *See* PENDATAHEADER data structure
PENINFO *See* PENINFO data structure
PENPACKET *See* PENPACKET data structure
RC *See* RC data structure
RCRESULT *See* RCRESULT data structure
RECTOFS *See* RECTOFS data structure
SKBINFO *See* SKBINFO data structure
STROKEINFO *See* STROKEINFO data structure
PFIELD 73
Debugging values 185
Defaults
BOXLAYOUT data structure 216
ink color 239
setting 194–95, 229
Delayed recognition
and compressed data 50
and scaled data 47
failure 256
sample code 70

DestroyPenData function 47, 111, 132
Device drivers
- *See also* Pen driver
- display drivers 2, 15, 45, 114
- modifying 105
- replacing 105

Dictionary
- .DLL 101, 102
- case statements 100
- constants 97–98
- definition 105
- description 17, 96–100, 114
- expanding functionality 239
- initializing 101–2
- languages 138
- loading 101–2
- maximum number of 97
- message groups 99
- path invalid 195
- processing
  - DIRQ_ subfunctions 101
  - fields 36, 96–99
  - functions 133–41
  - overview 22–23
  - sentences 35–36
  - words 35–36, 265
- sample code 138
- searches 142–43
- suggestions 35
- word lists 136–40

DictionaryProc function 96, 101, 133–41, 276
DictionarySearch function 98, 135, 142–43
DIRQ_ values
- DIRQ_ADD 99, 136, 138
- DIRQ_CLEANUP 99, 139
- DIRQ_CLOSE 99, 136, 138–39
- DIRQ_CONFIGURE 99, 136
- DIRQ_COPYRIGHT 99, 136, 139
- DIRQ_DELETE 99, 136, 138
- DIRQ_DESCRIPTION 99, 100, 137, 138–39
- DIRQ_FLUSH 99, 137
- DIRQ_INIT 99, 139
- DIRQ_OPEN 99, 137, 138–39, 140
- DIRQ_QUERY 99, 100, 133, 137, 138–39
- DIRQ_RCCHANGE 99, 137, 139, 140
- DIRQ_SETWORDLISTS 99, 138–39, 140
- DIRQ_STRING 98, 99, 100, 134, 138–39, 140
- DIRQ_SUGGEST 98, 99, 101, 134
- DIRQ_SYMBOLGRAPH 99, 134
- DIRQ_USER 99, 134

Display drivers 2, 15, 45, 114
.DLL
- dictionary 101, 138–40
- MAINDICT.DLL 96
- PENWIN 3, 15
- SHAPEREC 3
- USERDICT.DLL 96

DPtoTP function 110, 144
DrawArrow function 60
DrawPenData function 44, 111, 145
DRV_ messages
- DRV_GetCalibration 298–99
- DRV_GetName 298
- DRV_GetPenInfo 298–99
- DRV_GetVersion 298
- DRV_RemovePenDriverEntryPoints 297–98
- DRV_SetCalibration 298–99
- DRV_SetPenDriverEntryPoints 297–98
- DRV_SetPenSamplingDist 297–98
- DRV_SetPenSamplingRate 297–98

DuplicatePenData function 47, 111, 146
dwAppParam field 87, 239
dwDictParam field 87, 97, 99, 239
dwRecognizer field 87, 239
dwTick field 246

# E

Edit control 69, 192
Ellipse 53
EM_ values
- EM_GETSEL 296
- EM_LIMITTEXT 296
- EM_REPLACESEL 296
- EM_SETSEL 296

EmulatePen function 109, 147
EndEnumStrokes function 48, 111, 119, 148
EndPenCollection function 113, 149
EnumSymbols function 109, 150
Errors
- and corrections 205–10
- dictionary path invalid 195
- invalid parameters 195
- invalid recognition settings 195
- invalid user name 195
- segmentation 206

Events
- buffer 112
- generating pen 226
- mouse 172
- obtaining data 161–62
- pen into mouse/keyboard 15, 33

Events (*continued*)
processing queued pen events 179
recognizer processing 22
virtual 176–77
ExecuteGesture function 109, 151–52
EXPENSE.C 100
ExpenseDictionaryProc function 102–3

# F

Fields *See* individual field name
FirstSymbolFromGraph function 39, 109–10, 142, 153
fMinimized field 245
Fonts
in bedits 77
statements 77
Functions
allocation
GlobalAlloc 41, 47, 52
ANSI strings
CharacterToSymbol 121
application entry point
WinMain 54–55, 68
barrel button
GetPenAsyncState 156
bounding rectangles
BoundingRectFromPoints 120
buffer
CopyRawData function 60
GetPenHwData 93, 159–60
compression
CompactPenData 50, 123–24
custom recognizer
CloseRecognizer 89, 112
ConfigRecognizer 89, 112
InitRecognizer 89, 112
RecognizeDataInternal 89, 112
RecognizeInternal 89, 112
TrainContextInternal 89, 112
TrainInkInternal 89, 112
data access
GetPointsFromPenData 48, 163
data handling & display
DrawArrow function 60
DrawRawData function 60
DrawShape function 60
GetNearestColor 29
SetGraphWindow function 60
SetViewportOrg 145
SetWindowExt 145
SetWindowOrg 145

Functions (*continued*)
data I/O
AddPointsPenData 117, 207
EndPenCollection 149
dictionary
DictionaryProc 96, 101, 133–41, 276
DictionarySearch 98, 135, 142–43
GetProcAddress 23
LoadModule 23
events
IsPenAware 171
IsPenEvent 53, 155, 172
Keyboard_Event 24
ProcessPenEvent 116, 179
gestures
ExecuteGesture 151–52
GetSystemMetrics 295
hedit
CreateWindow 74
hooks
SetPenHook 196
SetRecogHook 197–98
HPENDATA data structure
BeginEnumStrokes 119
CreatePenData 207
GlobalLock 52
GlobalSize 52
initialization
InitRc 167–68, 229
ink
DrawPenData 44, 145
DuplicatePenData 47, 146
GetLPDevice 15
InkReady 15
keyboard
Keyboard_Event 24
ShowKeyboard 199–202
memory
AddPointsPenData 49
BeginEnumStrokes 48
CreatePenData 47
DestroyPenData 47, 132
EndEnumStrokes 48, 119, 148
GetPenDataInfo 48, 157
GlobalFree 47
SetClipboardData 295
mouse
GetMessageExtraInfo 20, 33, 155, 230
pen
AtomicVirtualEvent 118
EmulatePen 147

Functions (*continued*)
pen data
AddPointsPenData 111
BeginEnumStrokes 111
CompactPenData 111
CreatePenData 111
DestroyPenData 111
DrawPenData 111
DuplicatePenData 111
EndEnumStrokes 111
GetPenDataStroke function 111
GetPointsFromPenData 111
MetricScalePenData 111
OffsetPenData 111
ResizePenData 111
pen driver
description 113–14
UpdatePenInfo 213
pen module
AddPenEvent 113, 116
EndPenCollection 113
GetPenHWData 113
GetPenHWEventData 113
IsPenEvent 113
ProcessPenEvent 113
UpdatePenInfo 113
PENDATA data structure
CreatePenData 130–31
RC data structure defaults
GetGlobalRC 154, 289
SetGlobalRC 194–95, 289
recognition
CorrectWriting 109, 128–29
EmulatePen 109
ExecuteGesture 109
InitRC 109
InstallRecognizer 109, 169, 170
IsPenAware 109
lpFuncResults 93
ProcessWriting 5, 77, 80, 107, 109, 180–82, 282
Recognize 5, 21–22, 109, 183–85, 282
RecognizeData 9, 109, 186, 282
RecognizeDataInternal 93, 187
RecognizeInternal 22, 93–95, 188–89
RegisterPenApp 109
SetRecogHook 109
ShowKeyboard 109
UninstallRecognizer 109

Functions (*continued*)
recognizer
CloseRecognizer 122
ConfigRecognizer 125–27, 276
InitRecognizer 169
UninstallRecognizer 212
recognizer training
GetPenDataStroke 48, 119, 148, 158, 207
TrainContext 110, 205–6
TrainContextInternal 207–8
TrainInk 110, 205, 209–10
TrainInkInternal 211
rendering pen data
RedisplayPenData 45–46, 145, 190–91
stub
CloseRecognizer 93
ConfigRecognizer 92–93
InitRecognizer 92
TrainContext 92
TrainInkInternal 92
symbol graph
CalcNearestDir 95–96
EnumSymbols 150
FirstSymbolFromGraph 142, 153
GetSymbolCount 164
GetSymbolMaxLength 165
symbol manipulation
CharacterToSymbol 110
EnumSymbols 110
FirstSymbolFromGraph 39, 110
GetSymbolCount 110
GetSymbolMaxLength 110
SymbolToCharacter 39, 110, 203
tablet data
GetPenHwEventData 155, 161–62
transforming pen data
MetricScalePenData 46, 50, 173
OffsetPenData 43, 47, 175
ResizePenData 46, 193
utility
BoundingRectFromPoints 110
DPtoTP 110, 144
GetGlobalRC 110
GetMessageExtraInfo 110
GetPenAsyncState 110
GetVersionPenWin 110
SetGlobalRC 111
SetPenHook 111
TPtoDP 111, 204

Functions (*continued*)
 version number
  GetVersionPenWin 166
 virtual events
  AtomicVirtualEvent 109
  PostVirtualKeyEvent 109, 176
  PostVirtualMouseEvent 109, 177
 windows
  HformWndProc 68, 71
  InputWndProc 62–65
  MainWndProc 60–62
  RawWndProc 65
fVisible field 245

## G

Gesture
 accelerator 71
 ALC_ value 74
 and keyboard shortcuts 26
 binding 26
 definition 13
 executing 151–52
 gesture-only fields 29
 hot spots 85
 macro layer 24, 34
 manager 34, 39
 recognition process 29
GetGlobalRC function 110, 154, 289
GetLPDevice function 15
GetMessageExtraInfo function 20, 33, 110, 155, 230
GetNearestColor function 29
GetPenAsyncState function 110, 156
GetPenDataInfo function 48, 157
GetPenDataStroke function 48, 111, 119, 148, 158, 207
GetPenHwData function 93, 113, 159–60
GetPenHwEventData function 113, 155, 161–62
GetPointsFromPenData function 48, 111, 163
GetProcAddress function 23
GetSymbolCount function 110, 164
GetSymbolMaxLength function 109, 165
GetSystemMetrics function 295
GetVersionPenWin function 110, 166
GGRC_ values
 GGRC_DICTBUFTOOSMALL 154
 GGRC_OK 154
 GGRC_PARAMERROR 154
GlobalAlloc function 41, 47, 52
GlobalFree function 47
GlobalLock function 52
GlobalSize function 52
Graphs 37–38, 85–87, 98, 134–35
GUIDE data structure
 BXD_ values 255
 description 31, 216–17
 sample code 218
guide field 87, 167, 238

## H

Handwriting edit *See* Hedit
Handwriting processing 5, 6, 67–75, 180–82
HE_ messages
 HE_CHAROFFSET 277
 HE_CHARPOSITION 277
 HE_DEFAULTFONT 278
 HE_GETBOXLAYOUT 278
 HE_GETINFLATE 278
 HE_GETINKHANDLE 278
 HE_GETRC 278
 HE_GETRCRESULT 279
 HE_GETRCRESULTCODE 279
 HE_GETUNDERLINE 279
 HE_SETBOXLAYOUT 279
 HE_SETINFLATE 279
 HE_SETINKMODE 279
 HE_SETRC 280
 HE_SETUNDERLINE 280
 HE_STOPINKMODE 280
Hedit
 control creation code 72–75
 control messages 68, 277–81
 controls 69, 192
 description 4, 67–75
 setting hook after recognition 70
 setting RC preferences 74
 switch statement 74
 use 67
 window creation 74
HFORM parent window 71
HformWndProc function 68, 71
HKP_ values
 HKP_SETHOOK 197
 HKP_UNHOOK 197
HN_ messages
 HN_DELAYEDRECOGFAIL 256
 HN_ENDREC 256
 HN_RCRESULT 256
Hook
 pen packet 196
 recognition 197–98
Hot spots 85

HPENDATA data buffer
compressing data in 50
data points 42–43
description 41–52
freeing memory 132
generating 51
header 44
information storage 42
memory 46, 119
PENDATAHEADER structure 44
PENINFO structure 44
stroke headers 44
hpendata element 40
hpendata field 241
hrec field 32, 87, 168, 230
hwnd field 35, 88, 167, 230, 245, 281
HWR_ values
HWR_APPWIDE 197
HWR_RESULTS 197

# I

IDC_ values
IDC_ALTSELECT 257
IDC_PEN 257
.INI files
CONTROL.INI *See* CONTROL.INI file
format 285–86
PENWIN.INI *See* PENWIN
SYSTEM.INI *See* SYSTEM.INI file
Initialization
functions *See* Functions, initialization
data structures 70
dictionary 101–2
recognizer 169–71
SYSTEM.INI file 285–86
InitRC function 109, 167–68, 229
InitRecognizer function 89, 92, 112, 169
Ink
baseline 39
compressing 52
drawing process 21–22, 29
entering and storing 9
freeing data 35
functions 45–50
location and position 39
mapping of 38
midline 39
mode 18, 21–26
offsetting 51
processing 13, 77
redraw 190–91
Ink (*continued*)
rendering 45, 51
saving 42, 52
sizing 47, 52
width and color 29, 88, 145, 190, 239
Inking
definition 106
hidden cursor 34
restrictions 218
InkReady function 15
InstallRecognizer function 109, 169, 170
Installation requirements xii
IsPenAware function 109, 171
IsPenEvent function 53, 113, 155, 172

# K

Keyboard 199–202, 245
Keyboard_Event function 26
Keypad request
SKB_BASIC 200
SKB_FULL 200
SKB_NUMPAD 200

# L

Languages
dictionary 138
recognizing 32
sLanguage element in WIN.INI 33
word list codes 141
LoadModule function 23
lPcm field 88, 167, 237, 281
lPdc field 225–26
lpfnYield field 34, 88, 168, 231
lpFuncResults function 93
lpLanguage field 32, 88, 168, 232
lprc field 242
lprc pointer 40
lpsyv field 39, 241
lpUser field 35, 88, 168, 231
lRcOptions field 88, 97–98, 167, 231, 281

# M

Macros
IsAlcBitAnsi 235
layers 24, 34
ResetAlcBitAnsi 235
SetAlcBitAnsi 235
MAINDICT.DLL 96

Memory
adding data to 48
clearing 111
controlling functions 111
data structure description 223
freeing 132
housekeeping operations 47
HPENDATA data structure 41–42, 119
merging blocks in HPENDATA 46
operations 111
out of memory error 184, 270
retrieving data from 48
returning header & pen information 157
unlocking 148
Messages
bedit control 76
calibration driver 299
combo-box notification
CBN_DELAYEDRECOGFAIL 295
CBN_ENDREC 295
CBN_RCRESULT 295
control 68
dictionary
DIRQ_ADD 99
DIRQ_CLEANUP 99
DIRQ_CLOSE 99
DIRQ_CONFIGURE 99
DIRQ_COPYRIGHT 99
DIRQ_DELETE 99
DIRQ_DESCRIPTION 99
DIRQ_FLUSH 99
DIRQ_INIT 99
DIRQ_OPEN 99
DIRQ_QUERY 99
DIRQ_RCCHANGE 99
DIRQ_SETWORDLISTS 99
DIRQ_STRING 98, 99
DIRQ_SUGGEST 98, 99
DIRQ_SYMBOLGRAPH 99
DIRQ_USER 99
HN_ notification
HN_DELAYEDRECOGFAIL 256
HN_ENDREC 256
HN_RCRESULT 256
interpreter 15–16, 24
pen driver
DRV_GetCalibration 298–99
DRV_GetName 298
DRV_GetPenInfo 298–99
DRV_GetVersion 298
DRV_RemovePenDriverEntryPoints 297–98
DRV_SetCalibration 298–99
Messages (*continued*)
pen driver (*continued*)
DRV_SetPenDriverEntryPoints 297–98
DRV_SetPenSamplingDist 297–98
DRV_SetPenSamplingRate 297–98
results 84
SYV_
SYV_BACKSPACE 78
SYV_CLEAR 78
SYV_CLEARWORD 78
SYV_COPY 78
SYV_CORRECT 78
SYV_CUT 78
SYV_EXTENDSELECT 78
SYV_PASTE 78
SYV_RETURN 78
SYV_SPACE 78
SYV_TAB 78
SYV_UNDO 79
WM_
WM_GLOBALRCCHANGE 276
WM_RCRESULT 37, 282
WM_SKB 283
WM_HEDITCTL
HE_CHAROFFSET 277
HE_CHARPOSITION 277
HE_DEFAULTFONT 278
HE_GETBOXLAYOUT 278
HE_GETINFLATE 278
HE_GETINKHANDLE 278
HE_GETRC 278
HE_GETRCRESULT 279
HE_GETRCRESULTCODE 279
HE_GETUNDERLINE 279
HE_SETBOXLAYOUT 279
HE_SETINFLATE 279
HE_SETINKMODE 279
HE_SETRC 280
HE_SETUNDERLINE 280
HE_STOPINKMODE 280
MetricScalePenData function 46, 50, 111, 173
Microsoft recognizer 17
Microsoft Windows for Pen Computing
*See* Pen extensions
Module
Dictionary
DictionaryProc function 133–41
PenModule
AddPenEvent function 116
RC Manager
AddPointsPenData function 117
AtomicVirtualEvent function 118

Module (*continued*)
RC Manager (*continued*)
BeginEnumStrokes function 119
BoundingRectFromPoints function 120
CharacterToSymbol function 121
CompactPenData function 123–24
CorrectWriting function 128–29
CreatePenData function 130–31
DestroyPenData function 132
DictionarySearch function 142–43
DPtoTP function 144
DrawPenData function 145
DuplicatePenData function 146
EmulatePen function 147
EndEnumStrokes function 148
EndPenCollection function 149
EnumSymbols function 150
ExecuteGesture function 151–52
FirstSymbolFromGraph function 153
GetGlobalRC function 154
GetPenAsyncState function 156
GetPenDataInfo function 157
GetPenDataStroke function 158
GetPenHwData function 159–60
GetPenHwEventData function 161
GetPointsFromPenData function 163
GetSymbolCount function 164
GetSymbolMaxLength function 165
GetVersionPenWin function 166
InitRC function 167–68
InstallRecognizer function 170
IsPenAware function 171
IsPenEvent function 172
MetricScalePenData function 173
OffsetPenData function 175
PostVirtualKeyEvent function 176
PostVirtualMouseEvent function 177–78
ProcessPenEvent function 179
ProcessWriting function 180–82
Recognize function 183–85
RecognizeData function 186
SetGlobalRC function 194–95
SetPenHook function 196
SetRecogHook function 197–98
ShowKeyboard function 199–202
SymbolToCharacter function 203
TPtoDP function 204
TrainContext function 205–6
TrainInk function 209–10
UninstallRecognizer function 212
UpdatePenInfo funciton 213
Recognizer
CloseRecognizer function 122
ConfigRecognizer function 125–27
InitRecognizer function 169
RecognizeDataInternal function 187
RecognizeInternal function 190
RedisplayPenData function 190
RegisterPenApp function 192
ResizePenData function 193
TrainContextInternal function 207–8
TrainInkInternal function 211
Windows
GetMessageExtraInfo function 155
Mouse
click sample code 118
events 172
input code 286

## N

nBaseLine field 39
nInkWidth field 29, 88, 168, 239
nMidLine field 39
Noise reduction 82
Normalization 82
nPad field 245
nSamplingDist field 226
nSamplingRate field 226

## O

OCR 81, 112
OEMPENINFO data structure
description 82, 221–22
sample code 221–22
OffsetPenData function 43, 47, 111, 175
On-screen keyboard
command request
SKB_CENTER 200
SKB_HIDE 199
SKB_MINIMIZE 200
SKB_MOVE 200
SKB_QUERY 199
SKB_SHOW 199
partial keyboard 200
storage location 245

## P

Parameters, setting for recognizer 127
Parent window, HFORM 71

PCM_ values
 PCM_ADDDEFAULTS 238, 258
 PCM_INVERT 238, 258
 PCM_PENUP 29, 238, 258
 PCM_RANGE 29, 238, 258
 PCM_RECTBOUND 30, 40, 238, 258
 PCM_RECTEXCLUDE 30, 40, 238, 258
 PCM_TIMEOUT 30, 238, 258
PDC_ values
 PDC_BARREL1 225, 259
 PDC_BARREL2 225, 259
 PDC_BARREL3 225, 259
 PDC_INTEGRATED 225, 259
 PDC_INVERT 225, 259
 PDC_PROXIMITY 225, 259
 PDC_RANGE 225, 259
 PDC_RELATIVE 225, 259
PDK_ values
 PDK_BARREL1 156, 260
 PDK_BARREL2 156, 260
 PDK_BARREL3 156, 260
 PDK_DOWN 260
 PDK_DRIVER 260
 PDK_INVERTED 260
 PDK_OUTOFRANGE 260
 PDK_TRANSITION 260
PDT_ values
 PDT_ANGLEXY 221, 261
 PDT_ANGLEZ 221, 261
 PDT_BARRELROTATION 221, 261
 PDT_HEIGHT 221, 261
 PDT_NULL 221, 261
 PDT_OEMSPECIFIC 221, 261
 PDT_PRESSURE 221, 261
PDTS_ values
 PDTS_ARBITRARY 131, 223, 262
 PDTS_COMPRESS2NDDERIV 223
 PDTS_COMPRESSED 223
 PDTS_COMPRESSMETHOD 223
 PDTS_DISPLAY 173, 223, 262
 PDTS_HIENGLISH 130, 173, 223, 262
 PDTS_HIMETRIC 130, 173, 223, 262
 PDTS_LOMETRIC 130, 173, 224, 262
 PDTS_NOOEMDATA 224
 PDTS_NOPENINFO 224
 PDTS_NOUPPOINTS 224
 PDTS_SCALEMASK 224
 PDTS_SCALEMAX 224
 PDTS_STANDARDSCALE 131, 224, 262
 PDTS_TABLET 130, 262
PDTT_ trim options
 PDTT_ ALL 123
 PDTT_COLINEAR 123
 PDTT_COMPRESS 123
 PDTT_DECOMPRESS 124
 PDTT_DEFAULT 123
 PDTT_OEMDATA 124
 PDTT_PENINFO 124
 PDTT_UPPOINTS 124
Pen
 algorithms 46
 applications 16
 barrel button state 156
 determining last stroke 29
 determining recognition 29–30
 event 20, 226
 ink mode 18, 25–27
 input code 286
 interpreting strokes 81
 memory 157
 message interpreter 15–16, 24
 normal mode 18–19
 number per tablet 226
 obtaining data from buffer 159
 packet 116, 118, 179, 196, 228
 palette 118, 209
 stroke 44
 time-out period 30
 using to scroll 29
Pen data
 compression 50
 copying 146
 display 145
 housekeeping 47
 memory 47–48
 points 42–43
 saving 35, 264
 sizing 193
 structure 117
Pen driver
 communication with Windows 116, 228
 description 2, 14–15, 18
 functions 113–14
 IDC_ values 257
 installable 296
 messages 297–99
 PENINFO value changes 213
 sample code 296
 saving OEM data 221

Pen extensions
.DLL 3
components 14–17
data flow 17–26
description 1
device drivers 2
goals 13
objectives 1
pen interface 3
pen message interpreter 15–16, 24
recognizer 3
visual user feedback 34
PENAPP.C 90
PENDATA data structure 130
PENDATAHEADER data structure
description 44, 223–24
sample code 223–24
PENINFO data structure
description 44, 225–26
sample code 225
PENPACKET data structure
description 228
sample code 228
PENWIN
.DLL 3, 14
.INI file 195, 289–93
PFIELD data structure 73
pntEnd element 40
pntEnd field 241
PostVirtualKeyEvent function 109, 176
PostVirtualMouseEvent function 109, 177
Power requirements 7
Preferences
color 74
RC data structure field 230
setting for hedits 74
Process description *See* Data flow
ProcessPenEvent function 113, 116, 179
ProcessWriting function 5, 77, 80, 107, 109, 180–82, 282
Proximity detection 29

# R

RC data structure
bedit window class 31
description 5, 29–30, 229–30
dictionary code 96
filling in default values 167–68
GUIDE data structure 31
RC data structure (*continued*)
pointer to 40
sample code 229–30
setting defaults 194–95
RC data structure fields
alc 87, 167, 233
alcPriority 32, 87, 168, 234
clErrorLevel 87, 168, 233
dwAppParam 87, 239
dwDictParam 87, 97, 99, 239
dwRecognizer 87, 239
guide 87, 167, 238
hrec 32, 87, 168, 230
hwnd 88, 167, 230, 281
lPcm 88, 167, 237, 281
lpfnYield 88, 168, 231
lpLanguage 88, 168, 232
lpUser 88, 168, 231
lRcOptions 88, 97, 167, 231, 281
nInkWidth 29, 88, 168, 239
rectBound 30, 88, 167, 237, 281
rectExclude 30, 88, 167, 237
rgbfAlc 88, 168, 234
rgbInk 29, 88, 168, 239
rglpdf 53, 88, 97, 168, 232
rgwReserved 88, 239
wCountry 168, 232
wEventRef 88, 230, 281
wIntlPreferences 168, 232
wRcDirect 88, 167, 239
wRcOrient 88, 167, 239, 281
wRcPreferences 37, 88, 168, 231
wResultMode 89, 167, 236, 281
wTimeOut 30, 89, 168, 237
wTryDictionary 87, 97, 98, 168, 233
RC Manager
buffer 19, 20, 28
description 14, 89
interpreting pen events 20
RC preferences
color 74
setting for hedits 74
RCD_ values
RCD_BT 263
RCD_DEFAULT 263
RCD_LR 263
RCD_RL 263
RCD_TB 263
RCO_ values
RCO_BOXED 34, 264
RCO_COLDRECOG 264
RCO_DISABLEGESMAP 34, 264

RCO_ values (*continued*)
RCO_NOFLASHCURSOR 34, 264
RCO_NOFLASHUNKNOWN 34, 264
RCO_NOHIDECURSOR 34, 264
RCO_NOHOOK 35, 264
RCO_NOPOINTEREVENT 264
RCO_NOSPACEBREAK 35, 97–98, 264
RCO_SAVEALLDATA 35, 264
RCO_SAVEHPENDATA 35, 264
RCO_SUGGEST 35, 97–98, 265
RCO_TABLETCOORD 35, 265
RCOR_ values
RCOR_LEFT 266
RCOR_NORMAL 266
RCOR_RIGHT 266
RCOR_UPSIDEDOWN 266
RCP_ values
RCP_LEFTHAND 230, 267
RCP_MAPCHAR 231, 267
RCRESULT data structure
description 6, 36–40, 240–43
recognition 39
sample code 240
RCRT_ values
RCRT_ALREADYPROCESSED 242, 268
RCRT_DEFAULT 242
RCRT_GESTURE 242, 268
RCRT_GESTURETOKEYS 242, 268
RCRT_GESTURETRANSLATED 242, 268
RCRT_NORECOG 242, 268
RCRT_NOSYMBOLMATCH 243, 268
RCRT_PRIVATE 243, 268
RCRT_UNIDENTIFIED 243, 268
REC_ values
REC_ABORT 160, 183, 270
REC_ALC 185, 271
REC_BADEVENTREF 185, 270
REC_BADHPENDATA 184, 271
REC_BUFFERTOOSMALL 161, 184, 270
REC_BUSY 184, 270
REC_CLVERIFY 185, 271
REC_DEBUG 185, 271
REC_DICT 185, 271
REC_DONE 184, 270
REC_ERRORLEVEL 185, 271
REC_GUIDE 185, 271
REC_HREC 185, 271
REC_HWND 185, 271
REC_INVALIDREF 185, 271
REC_LANGUAGE 185, 272
REC_NOCOLLECTION 160, 185, 272
REC_NOINPUT 184, 270
REC_ values (*continued*)
REC_NOTABLET 184, 270
REC_OEM 185, 272
REC_OK 160, 161, 183, 270
REC_OOM 184, 270
REC_OVERFLOW 160, 184, 270
REC_PARAMERROR 160, 161
REC_PCM 185, 272
REC_POINTEREVENT 184, 270
REC_RECTBOUND 185, 272
REC_RECTEXCLUDE 185, 272
REC_RESULTMODE 185, 272
REC_TERMBOUND 160, 184, 271
REC_TERMEX 160, 184, 271
REC_TERMOEM 160, 184, 271
REC_TERMPENUP 160, 184, 271
REC_TERMRANGE 160, 184, 271
REC_TERMTIMEOUT 160, 184, 271
Recognition
alphabet 73
and writing direction order 31
bitmaps 81, 112
Context *See* RC data structure
controlling the process 33–35
data 81, 186–89, 230
delayed
and compressed data 50
and scaling pen data 46
definition 41
failure of 256
sample code for 70–71
description 22
ending 22, 29–30, 238
gesture manager 34
handwriting 6
hooks 35, 70
improving accuracy 32
languages 32
OCR 112
process description 28, 81–83
RC data structure 29–30
RCResult data structure 39
rectangles 30
results
character sets 84
description 36–40
errors plus corrections 205
gesture hot spots 85
message 84
optimum 220
overview 83

Recognition (*continued*)
- results (*continued*)
  - RCResult data structure
    - *See* RCResult data structure
  - storage space available 37
  - symbol graph 37–38
  - window they appear in 230
  - WM_RCRESULT message 37
- sentences 35
- shapes 53
- speed 84
- stroke order & direction 83
- tablet proximity detection 29
- time-out 35, 237
- timing of results 33
- vector 81, 112

Recognize functions
- Recognize 5, 21–22, 109, 183–85, 282
- RecognizeData 9, 109, 186, 282
- RecognizeDataInternal 89, 93, 112, 187
- RecognizeInternal 22, 89, 93–95, 112, 188–89

Recognizer
- and GUIDE structure 218
- call back function 231
- definition 17, 105
- description 89
- ending data collection 149
- event processing description 22
- functions 81
- handle 230
- initialization 169–71
- Microsoft 17
- removing 122, 212
- segmentation suggestions 206
- setting parameters 127
- training 85, 205–11
- using within a window 7
- yield function 231

rect field 245

Rectangles
- bounding 30, 39, 120, 167, 244
- recognition 30, 53
- sizing 47

rectBound field 30, 88, 167, 223, 237, 281

rectBoundInk field 39, 241

rectExclude field 30, 88, 167, 237

RECTOFS structure
- description 244
- sample code 244

RedisplayPenData function 45–46, 145, 190–91

Redraw ink 190–91

RegisterPenApp function 6, 69, 75, 109, 192

Removing recognizer 122, 212

ResizePenData function 47, 111, 193

Resizing data 193

Results *See* individual result name

rgbBox field 216

rgbfAlc field 88, 168, 234

rgbInk field 29, 88, 168, 239

rgbSelect field 216

rgbText field 216

rglpdf field 36, 53, 88, 97–98, 168, 232

rgoempeninfo field 227

rgwReserved field 88, 227, 239

RRM_ values
- RRM_COMPLETE 236
- RRM_NEWLINE 236
- RRM_STROKE 236
- RRM_SYMBOL 236
- RRM_WORD 235


# S


Sample code
- .INI files
  - CONTROL.INI 288
  - PENWIN.INI 289
  - SYSTEM.INI 286
- accelerator gesture 71
- adding pen events 116
- button bitmaps 201
- CalcNearestDir function 96
- calibration driver messages 299
- data structures
  - BOXLAYOUT 216
  - GUIDE 218
  - OEMPENINFO 221–22
  - PENDATAHEADER 223–24
  - PENPACKET 228
  - RC 229–30
  - RECTOFS 244
  - STROKEINFO 246
- dictionary 138
- EXPENSE.C 100
- hedit
  - and recognition hook 70
  - controls 69, 72–75
- mouse click 118
- Pen application 53–65
- PENAPP.C 53–65
- posting a mouse event 177
- RCRT_ values 268–69

Sample code (*continued*)
recognition
delayed 70
recognizer vs. RC manager 89–90
RecognizeInternal function 93–95
setting RC preferences 74
using ProcessWriting function 80
Sample dictionary 100–103
Saving
data points 35
ink data 52
pen data 35, 264
training results 208, 211
Scaling
data points 117
minimum & maximum 223
rectangles 47
Scrolling
implementing 29
in bedit control 76
Searches 142–43
Segmentation suggestions 206
SetClipboardData function 295
SetGlobalRC function 111, 194–95
SetGraphWindow function 60
SetPenHook function 111, 196
SetRecogHook function 109, 197–98
SetViewportOrg function 145
SetWindowExt function 145
SetWindowOrg function 145
SGRC_ values
SGRC_DICTIONARY 195
SGRC_INIFILE 195
SGRC_OK 195
SGRC_PARAMERROR 195
SGRC_RC 195
SGRC_RECOGNIZER 195
SGRC_USER 195
SHAPEREC.DLL 3
Shapes 10, 53
ShowKeyboard function 109, 199–202
Sizing data 193
SKB_ requests
SKB_BASIC 200
SKB_CENTER 200
SKB_FULL 200
SKB_HIDE 199
SKB_MINIMIZE 200
SKB_MOVE 200
SKB_NUMPAD 200
SKB_QUERY 199
SKB_SHOW 199
SKBINFO data structure
description 245
sample code 245
SKN_ values
SKN_MINCHANGED 283
SKN_PADCHANGED 283
SKN_POSCHANGED 283
SKN_VISCHANGED 283
Speed of recognition 84
Spell checking 135
Storage 7, 42
Strings
conversion to ANSI 203
converting from ANSI 121
count of 164
measure length of 165
sorting 150
Stroke
headers 44
recognition 83
STROKEINFO structure
description 246
points 49
sample code 246
stroke headers 43
style field 216
Subfunctions
ConfigRecognizer 125–27
CorrectWriting 128–29
Dictionary 133–35
SYC 38, 85, 247, 249
SYE 38, 85, 247–48
SYG 85, 248
syg field 241
Symbol
converting to characters 109–10
correspondence structures (SYC) 38, 85
elements (SYE) 38, 85
graphs 85–87, 98, 109–10, 134–35
values 85, 110
Symbol graph
description 37–38
interpreting 153
symbol count 164
SymbolToCharacter function 39, 110, 203
System
processing *See* Data flow
requirements xii

SYSTEM.INI file
description 285
modifying 286
mouse input code 286
pen tablet input code 286
SYV 247
SYV_ values
SYV_BACKSPACE 78, 181, 274
SYV_BEGINOR 273
SYV_CIRCLELOA 275
SYV_CIRCLELOZ 275
SYV_CIRCLEUPA 275
SYV_CIRCLEUPZ 275
SYV_CLEAR 78, 181, 274
SYV_CLEARWORD 78, 181, 274
SYV_COPY 78, 181, 274
SYV_CORRECT 78, 181, 274
SYV_CUT 78, 181, 274
SYV_EMPTY 273
SYV_ENDOR 273
SYV_EXTENDSELECT 78, 181, 274
SYV_NULL 273
SYV_OR 273
SYV_PASTE 78, 181, 274
SYV_RETURN 78, 181, 274
SYV_SOFTNEWLINE 274
SYV_SPACE 78, 181, 275
SYV_SPACENULL 274
SYV_TAB 78, 181, 275
SYV_UNDO 79, 182, 275
SYV_UNKNOWN 274
SYV_USER 275

# T

Tablets
coordinates 30, 35, 144, 204, 218
dimensions 226
hardware requirements 113
number of pens per 226
opaque digitizer 34
orientation 266
proximity detection 29
Terminating recognition process 29–30
Text
annotations 10
editing 128
spell checking 135
wrapping 76
TPtoDP function 111, 204
Trainable recognizers 85
TrainContext function 92, 110, 205–6
TrainContextInternal function 89, 112, 207–8
Training 205–11
TrainInk function 110, 205, 209–10
TrainInkInternal function 89, 92, 112, 211
Trim options
PDTT_ALL 123
PDTT_COLINEAR 123
PDTT_COMPRESS 123
PDTT_DECOMPRESS 124
PDTT_DEFAULT 123
PDTT_OEMDATA 124
PDTT_PENINFO 124
PDTT_UPPOINTS 124

# U

UninstallRecognizer function 109, 212
Unlocking memory 148
UpdatePenInfo function 113, 213
User
feedback 34
name 36
USERDICT.DLL 96

# V

Values *See* individual value names
Verifying results 23–24
Version number 166
Vertical scrolling 76
Virtual events 176–77
Visual user feedback 34
VWM_ constants
VWM_MOUSELEFTDOWN 177
VWM_MOUSELEFTUP 177
VWM_MOUSEMOVE 177
VWM_MOUSERIGHTDOWN 177
VWM_MOUSERIGHTUP 177

# W

wCountry field 168, 232
WCR_ subfunctions
WCR_CONFIGDIALOG 125
WCR_DEFAULT 125
WCR_PRIVATE 125
WCR_QUERY 125
WCR_QUERYLANGUAGE 126
WCR_RCCHANGE 126
WCR_RECOGNAME 126
WCR_TRAIN 126
WCR_TRAINCUSTOM 126

WCR_ subfunctions (*continued*)
  WCR_TRAINDIRTY 127
  WCR_TRAINMAX 127
  WCR_TRAINSAVE 127
  WCR_USERCHANGED 127
  WCR_VERSION 127
wDistinctHeight field 226
wDistinctWidth field 226
wEventRef field 88, 230, 281
WIN.INI file
  modifying 293–94
  PENWIN.INI 100
  sample code 293–94
  sLanguage element 32
Windows
  buffer 33
  communication with pen driver 116, 228
  creating 56, 70, 74
  hedit 68, 72–75
  HFORM parent 71
  invisible for interpreting messages 24
  setting to register pen entries 6
WinMain function 54–55, 70
wIntlPreferences field 168, 232
WM_ messages
  WM_GLOBALRCCHANGE 276
  WM_PASTE 296
  WM_RCRESULT 26, 37, 282
  WM_SKB 283
Word lists 136–40
wPdk field 246
wPndts field 44
wRcDirect field 31, 88, 167, 239
wRcOptions field 34
wRcOrient field 31, 88, 167, 239, 281
wRcPreferences field 35, 38, 88, 168, 231
wResultMode field 89, 167, 236, 281
wResultsType field 39, 241
Writing
  areas 8, 29–30
  directions 31, 239, 263
  on vertical axis 31
wTimeOut field 30, 89, 168, 237
wTryDictionary field 36, 87, 97, 98, 168, 233

## X

xOrigin field 218, 281

## Y

yOrigin field 218, 281

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-639

Microsoft®

0392 Part No. 28920